ARMv6-M Architecture Reference Manual

ARM®

ARM DDI 0419B_errata

ARMv6-M Architecture Reference Manual

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
March 2007	A	Non-Confidential	First release
September 2008	B	Non-Confidential, Restricted Access	Additions to the System Control Block, power management support, corrections to errata and clarifications
April 2009	B_errata	Non-Confidential	Errata to issue B, published only as a marked-up PDF document. See page iii for more information.

Proprietary Notice

Note

The term ARM is also used to refer to versions of the ARM architecture, for example ARMv6 refers to version 6 of the ARM architecture. The context makes it clear when the term is used in this way.

Note

- This errata PDF is regenerated from the source files of issue B of this document, but:
  - The pseudocode examples, that are inserted into the document, have been updated. Significant changes in the pseudocode are highlighted.
  - Pages ii and iii of the PDF have been replaced, by an edit to the PDF, to include an updated **Proprietary Notice**.

  With these exceptions, this PDF corresponds to the released PDF of issue C of the document, with errata indicated by markups to the PDF.

- Issues A and B of this Architecture Reference Manual described a number of features of the ARMv6-M architecture as comprising the OS Extensions. One effect of this errata PDF is to remove the definition of this extension. This means that:
  - the SP_process stack, the `SVC` instruction, and the PendSV exception become part of the baseline ARMv6-M architecture
  - the SysTick system timer and associated exception become an optional feature of the baseline ARMv6-M architecture.

Contents

ARMv6-M Architecture Reference Manual

List of Tables
ARMv6-M Architecture Reference Manual

List of Figures
ARMv6-M Architecture Reference Manual

Preface

This preface describes the contents of this manual, then lists the conventions and terminology it uses.

About this manual

This manual documents a substantially reduced version of the ARMv7-M Microcontroller profile. This architecture variant aligns strongly with the ARMv6 Thumb instruction set and is known as ARMv6-M. For short-form definitions of all the ARMv7 profiles and the ARMv6-M relationship see page A1-1.

The manual consists of three parts:

Part A The application level programming model and memory model information along with the instruction set as visible to the application programmer.

This is the information required to program applications or to develop the toolchain components (compiler, linker, assembler and disassembler) excluding the debugger.

Note

The ARM® architecture supports a common procedure calling standard, the ARM Architecture Procedure Calling Standard (AAPCS).

Part B The system level programming model and system level support instructions required for system correctness. The system level supports the ARMv6-M exception model. It also provides features for configuration and control of processor resources and management of memory access rights.

This is the information in addition to Part A required for an operating system (OS) and/or system support software. It includes details of register banking, the exception model, memory protection (management of access rights) and cache support.

Part B is profile specific. ARMv6-M and ARMv7-M share a new programmers' model and as such have some fundamental differences at the system level from the other ARM architecture profiles. As ARMv6-M is a memory-mapped architecture, the system memory map is documented here.

Part C The debug features to support the ARMv6-M debug architecture, and the programmer's interface to the debug environment.

This is the information required in addition to Parts A and B to write a debugger. Part C covers details of the different types of debug:

- halting debug and the related Debug state
- exception-based monitor debug
- non-invasive support for event generation and signalling of the events to an external agent.

This part is profile specific and includes several debug features unique within the architecture to the Microcontroller profile.

Using this manual

The information in this manual is organized into four parts, as described below.

Part A, Application level architecture

Part A describes the application level view of the architecture. It contains the following chapters:

Chapter A1 ***Introduction***

ARM architecture profile overview, background to the Microcontroller (M) profile, and the relationship between the ARMv6-M and ARMv7-M architecture profiles.

Chapter A2 ***Application Level Programmers' Model***

Details on the registers and status bits available at the application level along with a summary of the exception support.

Chapter A3 ***Application Level Memory Model***

Details of the ARM architecture memory attributes and memory order model.

Chapter A4 ***The ARMv6-M Instruction Set***

General instruction set information along with information on the different classifications of the instructions supported in ARMv6-M.

Chapter A5 ***Thumb Instruction Set Encoding***

Top level encoding details on bit field usage for the Thumb® instruction set support in ARMv6-M along with general details on UNDEFINED and UNPREDICTABLE terminology.

Chapter A6 ***Thumb Instruction Details***

Detailed reference material on each Thumb instruction, arranged alphabetically by instruction mnemonic. Summary information for system instructions is included and referenced for detailed definition in Part B.

Part B, System level architecture

Part B describes the system level view of the architecture. It contains the following chapters:

Chapter B1 ***System Level Programmers' Model***

Details of the registers, status and control mechanisms available at the system level.

Chapter B2 ***System Memory Model***

Details of the pseudocode used to support memory accesses to the ARM architecture memory model.

Chapter B3 ***System Address Map***

Overview of the system address map, and details of the architecturally defined features within the Private Peripheral Bus region. This chapter includes details of the memory-mapped support for a protected memory system.

Chapter B4 ***ARMv6-M System Instructions***

Contains detailed reference material on the system level instructions.

Part C, Debug architecture

Part C describes the Debug architecture. It contains the following chapter:

Chapter C1 ***ARMv6-M Debug***

ARMv6-M debug support.

Appendices

This manual contains a glossary and the following appendices:

Appendix A ***CoreSight Infrastructure IDs***

A summary of the ARM CoreSight™ compatible ID registers used for ARM architecture infrastructure identification.

Appendix B ***Legacy Instruction Mnemonics***

A cross reference of Unified Assembler Language forms of the instruction syntax to the Thumb format used in earlier versions of the ARM architecture.

Appendix C ***Deprecated Features in ARMv6-M***

Deprecated features that software is advised to avoid for future proofing. It is ARM's intent to remove this functionality in a future version of the ARM architecture.

Appendix D ***ARMv7-M Differences***

Key differences between the ARMv6-M and ARMv7-M Microcontroller profiles.

Appendix E ***Pseudocode definition***

Definition of terms, format and helper functions used by the pseudocode to describe the memory model and instruction operations.

Appendix F ***Pseudocode Index*** Index to definitions of pseudocode operators, keywords, functions, and procedures.

Appendix G ***Register Index*** Index to register descriptions in the manual.

Glossary

Glossary of terms - does not include terms associated with pseudocode.

Conventions

This manual employs typographic and other conventions intended to improve its ease of use.

General typographic conventions

`typewriter`	Is used for assembler syntax descriptions, pseudocode descriptions of instructions, and source code examples. For more details of the conventions used in assembler syntax descriptions see *Standard assembler syntax fields* on page A6-7. For more details of pseudocode conventions see Appendix E *Pseudocode definition*. The `typewriter` font is also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode descriptions of instructions and source code examples.
italic	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
bold	Is used for emphasis in descriptive lists and elsewhere, where appropriate.
SMALL CAPITALS	Are used for a few terms which have specific technical meanings.

Further reading

This section lists publications that provide additional information on the ARM architecture and ARM family of processors. This manual provides architecture information, the contract between hardware and software for development of ARM compliant cores, compiler and debug tools development and software to run on the ARM targets. The Technical Reference Manual (TRM) for the implementation of interest provides details of the IMPLEMENTATION DEFINED architecture features in the *ARM compliant core*. The silicon partner's device specification should be used for additional system details.

ARM periodically provides updates and corrections to its documentation. For the latest information and errata, some materials are published at `http://www.arm.com`. Alternatively, contact your distributor, or silicon partner who will have access to the latest published ARM information, as well as information specific to the device of interest. Your local ARM office has access to the latest published ARM information.

ARM publications

This document is specific to the ARMv6-M architecture. Other relevant publications relating to ARMv6-M and ARM's debug architecture are:

- *Procedure Call Standard for the ARM Architecture* (ARM GENC 003534)
- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *CoreSight Architecture Specification* (ARM IHI 0029)

For information on ARMv7-M, see the *ARMv7-M Architecture Reference Manual* (ARM DDI 0403).

For information on the ARMv7-A and -R profiles, see the *ARM Architecture Reference Manual* (ARM DDI 0406).

Feedback

ARM Limited welcomes feedback on its documentation.

Feedback on this book

If you notice any errors or omissions in this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Part A
Application Level Architecture

Chapter A1 Introduction

ARMv7 is documented as a set of architecture profiles. Three profiles have been defined as follows:

ARMv7-A	the application profile for systems supporting the ARM and Thumb instruction sets, and requiring virtual address support in the memory management model.
ARMv7-R	the realtime profile for systems supporting the ARM and Thumb instruction sets, and requiring physical address only support in the memory management model
ARMv7-M	the microcontroller profile for systems supporting only the Thumb instruction set, and where overall size and deterministic operation for an implementation are more important than absolute performance.

While profiles were formally introduced with the ARMv7 development, the A-profile and R-profile have implicitly existed in earlier versions, associated with the Virtual Memory System Architecture (VMSA) and Protected Memory System Architecture (PMSA) respectively.

ARMv6-M is a further development originally designed for FPGA implementations. It is a subset of ARMv7-M, providing a lightweight version of the Microcontroller profile's programming model, architecture extensions for ~~Operating System (OS) and~~ debug support, along with ARMv6 Thumb (16-bit) instruction set compatibility at the application level.

~~To support software development tools (compilers etc.), an ARMv6-M implementation with the OS Extension is defined as ARMv6S-M compliant. A summary of the OS Extension can be found in *Introduction to the system level* on page B1-2, and details of the~~ Debug Extension in Part C on page C1-1.

Note

ARMv6-M is upwardly compatible with ARMv7-M, meaning that application level and system level software developed for ARMv6-M can execute on ARMv7-M unmodified. ARMv7-M to ARMv6-M compatibility is not an architecture requirement. Many of the system level registers defined to support ARMv7-M features are reserved in ARMv6-M. Reserved registers exhibit UNK/SBZP behavior.

Instruction Set Architecture (ISA)

ARMv6-M supports the Thumb instruction set including a small number of 32-bit instructions introduced to the architecture as part of the Thumb-2 technology in ARMv6T2.

ARMv6-M supports the 16-bit Thumb instructions from ARMv7-M along with the 32-bit `BL`, `DMB`, `DSB`, `ISB`, `MRS` and `MSR` instructions.

Chapter A2
Application Level Programmers' Model

This chapter provides an application level view of the programmers' model. This is the information necessary for application development, as distinct from the system information required to service and support application execution under an operating system. It contains the following sections:

System related information is provided in overview form with references to the system information part of the architecture specification as appropriate.

A2.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system. That information is given in Chapter B1 *System Level Programmers' Model*.

A2.1.1 Privileged execution

System level support requires access to all features and facilities of the architecture, a level of access generally referred to as privileged operation. When an operating system supports both privileged and unprivileged operation, an application usually runs unprivileged. This:

- permits the operating system to allocate system resources to it in a unique or shared manner
- provides a degree of protection from other processes and tasks, and so helps protect the operating system from malfunctioning applications.

ARMv6-M only supports privileged operation. This means that an ARMv6-M application cannot be isolated from the underlying system code.

A2.1.2 System level architecture

Thread mode is the fundamental mode for application execution in ARMv6-M and is selected on reset. Thread mode can raise a supervisor call using the `SVC` instruction or handle system access and control directly.

All exceptions execute in Handler mode. Supervisor call (SVCall) handlers manage resources on behalf of the application such as interaction with peripherals, memory allocation and management of software stacks.

This chapter only provides a limited amount of system level information. Where appropriate it:

- gives an overview of the system level information
- gives references to the system level descriptions in Chapter B1 *System Level Programmers' Model* and elsewhere.

A2.2 ARM core data types and arithmetic

ARM processors support the following data types in memory:

Byte	8 bits
Halfword	16 bits
Word	32 bits
Doubleword	64 bits.

Processor registers are 32 bits in size. The Thumb instruction set contains instructions supporting the following data types held in registers:

- 32-bit pointers
- unsigned or signed 32-bit integers
- unsigned 16-bit or 8-bit integers, held in zero-extended form
- signed 16-bit or 8-bit integers, held in sign-extended form
- unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. You can load and store 64-bit integers using these instructions.

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to $2^{N}-1$, using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

Note

ARMv6-M has no direct instruction support for 64-bit integers.

A2.2.1 Integer arithmetic

The instruction set provides a wide variety of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and many others. These operations are defined using the *pseudocode* described in Appendix E *Pseudocode definition*, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in *Operators and built-in functions* on page AppxE-11.
- By use of pseudocode helper functions defined in the main text.
- By a sequence of the form:
    1. Use of the `SInt()`, `UInt()`, and `Int()` built-in functions defined in *Converting bitstrings to integers* on page AppxE-14 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
    2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
    3. Use of either the bitstring extraction operator defined in *Bitstring extraction* on page AppxE-12 or of the saturation helper functions described in *Pseudocode details of saturation* on page A2-9 to convert an unbounded integer result into a bitstring result that can be written to a register.

Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

(LSL) moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

(LSR) moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

(ASR) moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right (ROR) moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

(RRX) moves each bit of a bitstring right by one bit. The carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

ARMv6-M does not support the (RRX) operation.

Pseudocode details of shift and rotate operations

These shift and rotate operations are supported in pseudocode by the following functions:

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;

// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);

// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    if n == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, shift);
    return result;
```

Pseudocode details of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand(s) otherwise. For the precise definition of these operations, see *Addition and subtraction* on page AppxE-15.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. This status information can be used to synthesize multi-word additions and subtractions. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and carry and overflow outputs:

```
// AddWithCarry()
// ==============

(bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    signed_sum   = SInt(x) + SInt(y) + UInt(carry_in);
    result       = unsigned_sum<N-1:0>;  // == signed_sum<N-1:0>
    carry_out    = if UInt(result) == unsigned_sum then '0' else '1';
    overflow     = if SInt(result) == signed_sum then '0' else '1';
    return (result, carry_out, overflow);
```

An important property of the `AddWithCarry()` function is that if:

```
(result, carry_out, overflow) = AddWithCarry(x, NOT(y), carry_in)
```

then:

- If `carry_in == '1'`, then `result == x-y` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x >= y`).
- If `carry_in == '0'`, then `result == x-y-1` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x > y`).

Together, these mean that the `carry_in` and `carry_out` bits in `AddWithCarry()` calls can act as *NOT borrow* flags for subtractions as well as *carry* flags for additions.

Pseudocode details of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N. This is supported in pseudocode by the `SignedSatQ()` and `UnsignedSatQ()` functions when a boolean result is wanted saying whether saturation occurred, and by the `SignedSat()` and `UnsignedSat()` functions when only the saturated result is wanted:

Note

Saturation is not supported in ARMv6-M.

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);


// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);


// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;


// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

SatQ(i, N, unsigned) returns either UnsignedSatQ(i, N) or SignedSatQ(i, N) depending on the value of its third argument, and Sat(i, N, unsigned) returns either UnsignedSat(i, N) or SignedSat(i, N) depending on the value of its third argument:

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);



// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

A2.3 Registers and execution state

The application level programmers' model provides details of the general-purpose and special-purpose registers visible to the application programmer. Data is loaded into the registers from memory or stored from the registers to memory subject to the constraints of the ARM memory model (see Chapter A3 *Application Level Memory Model*). Data processing instructions affect data within the registers.

A2.3.1 ARM core registers

There are thirteen general-purpose 32-bit registers (R0-R12), and an additional three 32-bit registers that have special names and usage models.

SP stack pointer (R13), used as a pointer to the active stack. For usage restrictions see *Usage of 0b1101 as a register specifier* on page A5-3. This is preset to the top of the Main stack on reset. See *The SP registers* on page B1-8 for additional information.

LR link register (R14), used to store a value (the Return Link) relating to the return address from a subroutine that is entered using a Branch with Link instruction. The LR register is also updated on exception entry, see *Exception entry behavior* on page B1-18.

——— **Note** ———

R14 can be used for other purposes when the register is not required to support a return from a subroutine.

———————————

PC program counter (R15). For details on the usage model of the PC see *Use of 0b1111 as a register specifier* on page A5-2. The PC is loaded with the Reset handler start address on reset.

Pseudocode details of ARM core register operations

In pseudocode, the `R[]` function is used to:

- Read or write R0-R12, SP, and LR, using n == 0-12, 13, and 14 respectively.
- Read the PC, using n == 15.

This function has prototypes:

```
bits(32) R[integer n]
    assert n >= 0 && n <= 15;


R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
```

For more details on the `R[]` function, see *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-11. Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that, in ARMv6-M, must select the Thumb instruction set to execute after the branch.

Note

The following pseudocode defines behavior in ARMv6-M and the M-profile in general. It is much simpler than the equivalent pseudo-function definitions that apply to other ARM architecture profiles.

A simple branch is performed by the `BranchWritePC()` function:

```
// BranchWritePC()
// ==============

BranchWritePC(bits(32) address)
    BranchTo(address<31:1>:'0');
```

BLXWritePC is now a separate pseudofunction in the M-profile.
See note for details:

An interworking branch is performed by the `BXWritePC()` function:

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if CurrentMode == Mode_Handler && address<31:28> == '1111' then
        ExceptionReturn(address<27:0>);
    else
        EPSR.T = address<0>;  // if EPSR.T == 0, a HardFault
                              // is taken on the next instruction
        BranchTo(address<31:1>:'0');
```

The `LoadWritePC()` and `ALUWritePC()` functions are used for two cases where the behavior was systematically modified between architecture versions. The functions simplify to aliases of the branch functions in the M-profile architecture variants:

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address);

// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    BranchWritePC(address);
```

A2.3.2 The Application Program Status Register (APSR)

Program status is reported in the 32-bit Application Program Status Register (APSR), where the defined bits break down into a set of flags as follows:

31	30	29	28	27 … 0
N	Z	C	V	Reserved

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:

N, bit [31]	Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.
Z, bit [30]	Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
C, bit [29]	Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
V, bit [28]	Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

A2.3.3 Execution state support

ARMv6-M only executes Thumb instructions, and therefore always executes instructions in Thumb state. See Chapter A6 *Thumb Instruction Details* for a list of the instructions supported.

In addition to normal program execution, a Debug state exists when the Debug Extension is implemented – see Chapter C1 *ARMv6-M Debug* for more details.

A2.4 Exceptions, faults and interrupts

An exception can be caused by the execution of an exception generating instruction or triggered as a response to a system behavior such as an interrupt, alignment fault or memory system fault. Synchronous and asynchronous exceptions can occur within the architecture.

A2.4.1 System related events

The following types of exception are system related. Where there is direct correlation with an instruction, reference to the associated instruction is made.

Supervisor calls are used by application code to request a service from the underlying operating system.

Several forms of Fault can occur:

- instruction execution related errors
- data memory access errors on any load or store
- usage faults from a variety of execution state related errors, for example execution of an UNDEFINED instruction.

Faults in general are synchronous with respect to the associated executing instruction. Some system errors can cause an imprecise exception where it is reported at a time bearing no fixed relationship to the instruction which caused it.

Interrupts are always treated as asynchronous events with respect to the program flow. System timer (SysTick), a Pending[1] service call (PendSV[2]), and a controller for external interrupts (NVIC) are all defined. See *System timer - SysTick* on page B3-13 for information on the SysTick interrupt, and *Nested Vectored Interrupt Controller (NVIC)* on page B3-17 for information on the interrupt controller. PendSV is supported by the Interrupt Control State register (see *Interrupt Control State Register (ICSR)* on page B3-8).

A BKPT instruction generates a debug event if the Debug Extension is supported and enabled – see *Debug event behavior* on page C1-10 for more information.

1. For the definition of a Pending exception, see *Exceptions* on page B1-4.
2. A service (system) call is used by an application which requires a service from an underlying operating system. The service call associated with PendSV executes when the interrupt is taken. For a service call which executes synchronously with respect to program execution use the SVC instruction.

A2.5 Coprocessor support

ARMv6-M does not support coprocessors.

Chapter A3
Application Level Memory Model

This chapter covers the general principles which apply to the ARM memory model. The chapter contains the following sections:

ARMv6-M is a memory-mapped architecture. The address map specific details that apply to ARMv6-M are described in *The system address map* on page B3-2.

A3.1 Address space

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to 2^{32} - 1.

This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3. The address space can also be considered as consisting of 2^{31} 16-bit halfwords, each of whose addresses is halfword-aligned, which means that the address is divisible by 2. The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

For ARMv6-M, instruction fetches are always halfword-aligned and data accesses are always naturally aligned.

Address calculations are normally performed using ordinary integer instructions. This means that they wrap around if they overflow or underflow the address space. Another way of describing this is that any address calculation is reduced modulo 2^{32}.

Normal sequential execution of instructions effectively calculates:

```
(address_of_current_instruction) +(2 or 4)  /*16- and 32-bit instruction mix*/
```

after each instruction to determine which instruction to execute next. If this calculation overflows the top of the address space, the result is UNPREDICTABLE. In ARMv6-M this condition cannot occur because the top of memory is defined to always have the eXecute Never (XN) memory attribute associated with it. See *The system address map* on page B3-2 for more details. An access violation will be reported if this scenario occurs.

The above only applies to instructions that are executed, including those which fail their condition code check. Most ARM implementations prefetch instructions ahead of the currently-executing instruction.

`LDM`, `POP`, `PUSH`, and `STM` instructions access a sequence of words at increasing memory addresses, effectively incrementing a memory address by 4 for each register load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE.

A3.1.1 Virtual versus physical addressing

Virtual memory is not supported in ARMv6-M. A virtual address (VA) is always equal to a physical address (PA).

A3.2 Alignment support

ARMv6-M always generates a fault when an unaligned access occurs.

Writes to the PC are restricted according to the rules outlined in *Use of 0b1111 as a register specifier* on page A5-2.

A3.2.1 Alignment behavior

Address alignment affects data accesses and updates to the PC.

Alignment and data access

The following data accesses always generate an alignment fault:

- Non word-aligned `LDMIA` and `POP`
- Non word-aligned `STMIA and PUSH`
- Non halfword-aligned `LDR{S}H` and `STRH`
- Non word-aligned `LDR` and `STR`.

Alignment and updates to the PC

All instruction fetches must be halfword-aligned.

For exception entry and return:

- exception entry using a vector with bit [0] clear causes a HardFault[1]
- execution of a reserved EXC_RETURN is UNPREDICTABLE
- loading an unaligned value from the stack into the PC on an exception return is UNPREDICTABLE.

For all other cases where the PC is updated:

- bit [0] of the value is ignored when loading the PC using an `ADD` or `MOV` instruction
- a `BLX`, `BX`, or `POP` (including the PC) instruction will cause a HardFault[1] if bit [0] of the value written to the PC is zero.

1. The vector generates an invalid execution state that causes a HardFault when the first instruction of the exception handler is executed.

A3.3 Endian support

The address space rules (*Address space* on page A3-2) require that for an address A:

- the word at address A consists of the bytes at addresses A, A+1, A+2 and A+3
- the halfword at address A consists of the bytes at addresses A and A+1
- the halfword at address A+2 consists of the bytes at addresses A+2 and A+3
- the word at address A therefore consists of the halfwords at addresses A and A+2.

However, this does not fully specify the mappings between words, halfwords and bytes. A memory system uses one of the following mapping schemes. This choice is known as the endianness of the memory system.

In a *little-endian* memory system the mapping between bytes from memory and the interpreted value in an ARM register is illustrated in Table A3-1.

- a byte or halfword at address A is the least significant byte or halfword within the word at that address
- a byte at a halfword address A is the least significant byte within the halfword at that address.

Table A3-1 Little-endian byte format

	31 24	**23 16**	**15 8**	**7 0**
Word at Address A	Byte {Addr + 3}	Byte {Addr + 2}	Byte {Addr + 1}	Byte {Addr + 0}
Halfword at Address A			Byte {Addr + 1}	Byte {Addr + 0}

In a *big-endian* memory system the mapping between bytes from memory and the interpreted value in an ARM register is illustrated in Table A3-2.

- a byte or halfword at address A is the most significant byte or halfword within the word at that address
- a byte at a halfword address A is the most significant byte within the halfword at that address.

Table A3-2 Big-endian byte format

	31 24	**23 16**	**15 8**	**7 0**
Word at Address A	Byte {Addr + 0}	Byte {Addr + 1}	Byte {Addr + 2}	Byte {Addr + 3}
Halfword at Address A			Byte {Addr + 0}	Byte {Addr + 1}

For a word address A, Table A3-3 and Table A3-4 show how the word at address A, the halfwords at address A and A+2, and the bytes at addresses A, A+1, A+2 and A+3 map onto each other for each endianness.

Table A3-3 Little-endian memory system

MSByte	MSByte -1	LSByte + 1	LSByte
Word at Address A			
Halfword at Address A+2		Halfword at Address A	
Byte at Address A+3	Bye at Address A+2	Byte at Address A+1	Byte at Address A

Table A3-4 Big-endian memory system

MSByte	MSByte -1	LSByte + 1	LSByte
Word at Address A			
Halfword at Address A		Halfword at Address A+2	
Byte at Address A	Byte at Address A+1	Byte at Address A+2	Byte at Address A +3

The big-endian and little-endian mapping schemes determine the order in which the bytes of a word or half-word are interpreted.

As an example, a load of a word (4 bytes) from address `0x1000` will result in an access of the bytes contained at memory locations `0x1000`, `0x1001`, `0x1002` and `0x1003`, regardless of the mapping scheme used. The mapping scheme determines the significance of those bytes.

A3.3.1 Control of the Endian Mapping in ARMv6-M

It is IMPLEMENTATION DEFINED in ARMv6-M whether the selection of big endian (BE) or little endian (LE) memory mapping is a build time option or determined from a control input on a reset. The endian mapping has the following restrictions:

- The endian setting only applies to data accesses, instruction fetches are always little endian
- Loads and stores to the Private Peripheral Bus (*General rules applying to PPB register access* on page B3-4) are always little endian

For information on endian control and status see *Control of the Endian Mapping in ARMv6-M* on page B2-8.

Instruction alignment and byte ordering

ARMv6-M enforces 16-bit alignment on all instructions. This means that 32-bit instructions are treated as two halfwords, hw1 and hw2, with hw1 at the lower address.

In instruction encoding diagrams, hw1 is shown to the left of hw2. This results in the encoding diagrams reading more naturally. The byte order of a 32-bit Thumb instruction is shown in Figure A3-1.

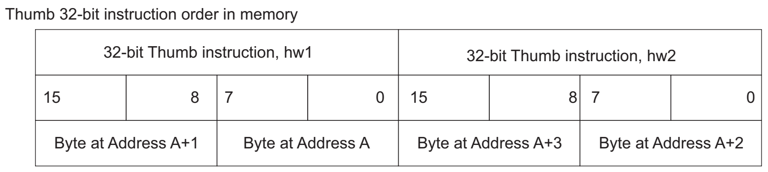
Thumb 32-bit instruction order in memory
32-bit Thumb instruction, hw1
32-bit Thumb instruction, hw2
15
8
7
0
15
8
7
0
Byte at Address A+1
Byte at Address A
Byte at Address A+3
Byte at Address A+2

Figure A3-1 Instruction byte order in memory

A3.3.2 Element size and Endianness

The effect of the endianness mapping on data applies to the size of the element(s) being transferred in the load and store instructions. Table A3-5 shows the element size of each of the load and store instructions:.

Table A3-5 Load-store and element size association

Instruction class	Instructions	Element Size
Load/store byte	`LDR{S}B, STRB`	byte
Load/store halfword	`LDR{S}H, STRH`	halfword
Load/store word	`LDR, STR`	word
Load/store multiple words	`LDM{IA}, STM{IA}, PUSH, POP`	word

A3.3.3 Instructions to reverse bytes in a general-purpose register

When an application or device driver has to interface to memory-mapped peripheral registers or shared-memory structures that are not the same endianness as that of the internal data structures, or the endianness of the Operating System, an efficient way of being able to explicitly transform the endianness of the data is required.

ARMv6-M provides instructions for the following byte transformations (see the instruction definitions in Chapter A6 *Thumb Instruction Details* for details):

`REV`	Reverse word (four bytes) register, for transforming 32-bit representations.
`REVSH`	Reverse halfword and sign extend, for transforming signed 16-bit representations.
`REV16`	Reverse packed halfwords in a register for transforming unsigned 16-bit representations.

A3.4 Synchronization and semaphores

Exclusive access instructions support non-blocking shared-memory synchronization primitives that allow calculation to be performed on the semaphore between the read and write phases, and scale for multi-processor system designs.

Note

ARMv6-M does not support exclusive accesses.

A3.5 Memory types

The ARM architecture defines a set of memory attributes with the characteristics required to support all memory and devices in the system memory map. The ordering of accesses for regions of memory is also defined by the memory attributes.

There are three mutually exclusive main memory type attributes to describe the memory regions:

- Normal
- Device
- Strongly Ordered.

Memory used for program execution and data storage generally complies with Normal memory. Examples of Normal memory technology are:

- preprogrammed Flash (updating Flash memory can impose stricter ordering rules)
- ROM
- SRAM
- SDRAM and DDR memory.

System peripherals (I/O) generally conform to different access rules; defined as Strongly Ordered or Device memory. Examples of I/O accesses are:

- FIFOs where consecutive accesses add (write) or remove (read) queued values
- interrupt controller registers where an access can be used as an interrupt acknowledge changing the state of the controller itself
- memory controller configuration registers that are used to set up the timing (and correctness) of areas of Normal memoryNormal memory
- memory-mapped peripherals where the accessing of memory locations causes side effects within the system.

In addition, the *Shareable* attribute indicates whether Normal or Device memory is private to a single processor, or accessible from multiple processors or other bus master resources, for example, an intelligent peripheral with DMA capability.

Strongly Ordered memory is required where it is necessary to ensure strict ordering of the access with respect to what occurred in program order before the access and after it. Strongly Ordered memory always assumes the resource is Shareable.

Table A3-6 provides a summary of the memory attributes.

Table A3-6 Summary of memory attributes

Memory type attribute	Shareable attribute	Other attribute	Description
Strongly ordered			All memory accesses to Strongly Ordered memory occur in program order. All Strongly Ordered accesses are assumed to be Shareable.
Device	Shareable		Designed to handle memory mapped peripherals that are shared by several processors.
	Non-Shareable		Designed to handle memory mapped peripherals that are used only by a single processor.
Normal	Shareable	Non-cacheable/Write-Through cacheable/Write-Back cacheable	Designed to handle Normal memory which is shared between several processors.
	Non-Shareable	Non-cacheable/Write-Through cacheable/Write-Back cacheable	Designed to handle Normal memory which is used only by a single processor.

A3.5.1 Atomicity

The terms *Atomic* and *Atomicity* are used within computer science to describe a number of properties for memory accesses. Within the ARM architecture, the following definitions are used:

Single-copy atomicity

The property of *Single-copy atomicity* is exhibited for read and write operations if the following conditions are met:

1. After every two write operations to an operand, either the value of the first write operation or the value of the second write operation remains in the operand. Thus, it is impossible for part of the value of the first write operation and part of the second write operation to remain in the operand.

2. When a read operation and a write operation occur to the same operand, the value obtained by the read operation is either the value of the operand before the write operation or the value of the operand after the write operation. It is never the case that the value of the read operation is partly the value of the operand before the write operation and partly the value of the operand after the write operation.

The only ARMv6-M explicit accesses made by the ARM processor which exhibit single-copy atomicity are:

- all byte transactions
- all halfword transactions to 16-bit aligned locations
- all word transactions to 32-bit aligned locations.

LDM, STM, PUSH and POP operations are seen to be a sequence of 32-bit transactions aligned to 32 bits. Each of these 32-bit transactions are guaranteed to exhibit single-copy atomicity. Sub-sequences of two or more 32-bit transactions from the sequence might not exhibit single-copy atomicity.

When an access is not single-copy atomic, it is executed as a sequence of smaller accesses, each of which is single-copy atomic, at least at the byte level.

For implicit accesses:

- Cache linefills and evictions have no effect on the atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic for each instruction fetched.

Note

32-bit thumb instructions are fetched as two 16-bit items.

Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *Multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all copies of the location.
- A read of a location does not return the value of a write until all copies of the location have seen that write.

Writes to Normal memory are not multi-copy atomic.

All writes to Device and Strongly-Ordered memory that are single-copy atomic are also multi-copy atomic.

All write accesses to the same location are serialized. Write accesses to Normal memory can be repeated up to the point that another write to the same address is observed.

For Normal memory, serialization of writes does not prohibit the merging of writes.

A3.5.2 Normal memory attribute

Normal memory is idempotent, exhibiting the following properties:

- read transactions can be repeated with no side effects
- repeated read transactions return the last value written to the resource being read

- read transactions can prefetch additional memory locations with no side effects
- write transactions can be repeated with no side effects, provided that the location is unchanged between the repeated writes
- unaligned accesses can be supported
- transactions can be merged prior to accessing the target memory system.

Normal memory can be read/write or read-only. The Normal memory attribute can be further defined as being Shareable or Non-Shareable, and describes most memory used in a system.

Accesses to Normal memory conform to the weakly-ordered model of memory ordering. A description of the weakly-ordered model can be found in standard texts describing memory ordering issues. A recommended text is chapter 2 of Memory Consistency Models for Shared Memory-Multiprocessors, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685.

All explicit accesses must correspond to the ordering requirements of accesses described in *Memory access order* on page A3-16.

Instructions which conform to the sequence of transactions classification as defined in *Atomicity* on page A3-9 can be abandoned if an exception occurs during the sequence of transactions. The instruction will be restarted on return from the exception, and one or more of the memory locations can be accessed multiple times. For Normal memory, this can result in repeated write transactions to a location which has been changed between the repeated writes.

Non-Shareable Normal memory

The Non-Shareable Normal memory attribute is designed to describe Normal memory that can be accessed only by a single processor.

A region of memory marked as Non-Shareable Normal does not have any requirement to make the effect of a cache transparent. For regions of memory marked as Non-Shareable Non-cacheable, a Data Synchronization Barrier (DSB) instruction must be used in situations where previous accesses must be made visible to other observers. See *Memory barriers* on page A3-19 for more details.

Shareable Normal memory

The Shareable Normal memory attribute is designed to describe Normal memory that can be accessed by multiple processors or other system masters.

A region of memory marked as Shareable Normal is one in which the effect of interposing a cache (or caches) on the memory system is entirely transparent to data accesses. Explicit software management is still required to ensure coherency of instruction caches. Implementations can use a variety of mechanisms to support this, from very simply not caching accesses in Shareable regions to more complex hardware schemes for cache coherency for those regions.

Cacheable write-through, cacheable write-back and non-cacheable memory

In addition to marking a region of Normal memory as being Shareable or Non-Shareable, regions can also be marked as being one of:

- cacheable write-through
- cacheable write-back
- non-cacheable.

This marking is independent of the marking of a region of memory as being Shareable or Non-Shareable. It indicates the required handling of the data region for reasons other than those to handle the requirements of shared data. As a result, it is acceptable for a region of memory that is marked as being cacheable and shareable not to be held in the cache in an implementation which handles shared regions by not caching the data.

A3.5.3 Device memory attribute

The Device memory attribute is defined for memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Memory mapped peripherals and I/O locations are typical examples of areas of memory that should be marked as being Device.

Explicit accesses from the processor to regions of memory marked as Device occur at the size and order defined by the instruction. The number of accesses that occur to such locations is the number that is specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not restartable. An example where an implementation might want to repeat an access is before and after an interrupt, in order to allow the interrupt to cause a slow access to be abandoned. Such implementation optimizations must not be performed for regions of memory marked as Device.

In addition, address locations marked as Device are non-cacheable. While writes to device memory can be buffered, writes shall only be merged where the correct number of accesses, order, and their size is maintained. Multiple accesses to the same address cannot change the number or order of accesses to that address. Coalescing of accesses is not permitted in this case.

Accesses to Device memory can exhibit side effects. Device memory operations that have side effects that apply to Normal memory locations require Memory Barriers to ensure correct execution. An example is the programming of the configuration registers of a memory controller with respect to the memory accesses it controls.

All explicit accesses to memory marked as Device must correspond to the ordering requirements of accesses described in *Memory access order* on page A3-16.

Shareable attribute

The Shareable attribute is defined by memory region and can be referred to as:

- memory marked as Shareable Device
- memory marked as Non-Shareable Device

Memory marked as Non-Shareable Device is defined as only accessible by a single processor. An example of a system supporting Shareable and Non-Shareable Device memory is an implementation that supports a local bus for its private peripherals, whereas system peripherals are situated on the main (Shareable) system bus. Such a system can have more predictable access times for local peripherals such as watchdog timers or interrupt controllers.

A3.5.4 Strongly Ordered memory attribute

Accesses to memory marked as Strongly Ordered have a strong memory-ordering model for all explicit memory accesses from that processor. An access to memory marked as Strongly Ordered is required to act as if a DMB memory barrier instruction were inserted before and after the access from that processor. See *Data Memory Barrier (DMB)* on page A3-19 for DMB details.

Explicit accesses from the processor to memory marked as Strongly Ordered occur at their program size, and the number of accesses that occur to such locations is the number that are specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not restartable.

Address locations marked as Strongly Ordered are not held in a cache, and are always treated as Shareable memory locations.

All explicit accesses to memory marked as Strongly Ordered must correspond to the ordering requirements of accesses described in *Memory access order* on page A3-16.

A3.5.5 Memory access restrictions

The following restrictions apply to memory accesses:

- For any access X, the bytes accessed by X must all have the same memory type attribute, otherwise, the behavior of the access is UNPREDICTABLE. That is, unaligned[1] accesses that span a boundary between different memory types are UNPREDICTABLE.
- For any two memory accesses X and Y, such that X and Y are generated by the same instruction, X and Y must all have the same memory type attribute, otherwise, the results are UNPREDICTABLE. For example, an `LDM` or `STM` that spans a boundary between Normal and Device memory is UNPREDICTABLE.
- Instructions that generate unaligned[1] memory accesses to Device or Strongly Ordered memory are UNPREDICTABLE.
- For instructions that generate accesses to Device or Strongly Ordered memory, implementations must not change the sequence of accesses specified by the pseudocode of the instruction. This includes not changing:
  - — how many accesses there are
  - — the time order of the accesses
  - — the data sizes and other properties of each access.

1. In ARMv6-M an unaligned access causes a HardFault rather than UNPREDICTABLE behavior.

In addition, processor implementations expect any attached memory system to be able to identify the memory type of an access, and to obey similar restrictions with regard to the number, time order, data sizes and other properties of the access.

Exceptions to this rule are:

- An implementation of a processor can break this rule, provided that the information it does supply to the memory system enables the original number, time order, and other details of the accesses to be reconstructed. In addition, the implementation must place a requirement on attached memory systems to do this reconstruction when the accesses are to Device or Strongly Ordered memory.

  For example, the word loads generated by an `LDM` can be paired into 64-bit accesses by an implementation with a 64-bit bus. This is because the instruction semantics ensure that the 64-bit access is always a word load from the lower address followed by a word load from the higher address, provided a requirement is placed on memory systems to unpack the two word loads where the access is to Device or Strongly Ordered memory.
- Any implementation technique that produces results that cannot be observed to be different from those described above is legitimate.

- In ARMv6-M, it is IMPLEMENTATION DEFINED if interrupts are taken during the execution of a multi-word instruction (`LDM`, `STM`, `PUSH` or `POP`). Memory accesses might repeat if a multi-word instruction is restarted after an exception, therefore ARM recommends that multi-word instructions are not used to access memory marked as Device or Strongly Ordered. For details on the architecture constraints associated with LDM and STM and the exception model see *Exceptions in LDM and STM operations* on page B1-24.

- Multi-access instructions that load or store the PC must only access Normal memory. If they access Device or Strongly Ordered memory the results are UNPREDICTABLE.

- Instruction fetches must only access Normal memory. If they access Device or Strongly Ordered memory, the results are UNPREDICTABLE. For example, instruction fetches must not be performed to areas of memory containing read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses.

To ensure correctness, read-sensitive locations must be marked as non-executable (see *User/privileged access and Read/Write access control for Instruction Accesses* on page A3-15). In ARMv6-M accessibility is fixed as part of the memory map, see *The system address map* on page B3-2 for more details.

A3.6 Access rights

Access rights split into two classes:

- Rights for data accesses
- Rights for instruction prefetching

Furthermore, the access right can be restricted to privileged execution only.

A3.6.1 Privileged access and Read/Write access control for Data Accesses

The ARM architecture memory attributes are allowed to define for explicit reads and for explicit writes that a region of memory is:

- not accessible to any accesses
- accessible only to Privileged accesses
- accessible to Privileged and Unprivileged accesses.

Note

ARMv6-M accesses are always privileged.

Not all combinations of memory attributes for reads and writes are supported by all systems which define the memory attributes.

A3.6.2 User/privileged access and Read/Write access control for Instruction Accesses

The ARM architecture memory attributes can define that a region of memory is:

- not accessible for execution - prefetching must not occur from locations marked as non-executable
- accessible for execution by Privileged processes only
- accessible for execution by Privileged and Unprivileged processes.

Note

ARMv6-M accesses are always privileged.

The mechanism by which this is described is that the region is described as accessible for reads by a privileged read access (or by privileged and unprivileged read access) and is suitable for execution. As a result, there is some linkage between the memory attributes which define the accessibility to explicit memory accesses, and those which define that a region can be executed.

If execution is attempted from any memory location which the attributes prohibit, an instruction execution error will occur.

A3.7 Memory access order

The memory types defined in *Memory types* on page A3-8 have associated memory ordering rules to provide system compatibility for software between different implementations. The rules are defined to accommodate the increasing difficulty of ensuring linkage between the completion of memory accesses and the execution of instructions within a complex high-performance system, while also allowing simple systems and implementations to meet the criteria with predictable behavior.

The memory order model determines:

- when side effects are guaranteed to be visible
- the requirements for memory consistency

Shareable memory indicating whether a region of memory is shared between multiple processors (and therefore requires an appearance of cache transparency in an ordering model) is supported. Implementations remain free to choose the mechanisms to implement this functionality.

Additional attributes and behaviors relate to the memory system architecture. These features are defined in other areas of this manual (see *Access rights* on page A3-15 and Table B3-1 on page B3-3 on access permissions and the system memory map respectively).

A3.7.1 Read and write definitions

Memory accesses can be either reads or writes.

Reads

Reads are defined as memory operations that have the semantics of a load. For ARMv6-M and Thumb these are:

- `LDR{S}B`, `LDR{S}H`, `LDR`
- `LDM{IA}`, `POP`

Writes

Writes are defined as operations that have the semantics of a store. For ARMv6-M and Thumb these are:

- `STRB`, `STRH`, `STR`
- `STM{IA}`, `PUSH`

Memory synchronization primitives

Synchronization primitives are required to ensure correct operation of system semaphores within the memory order model.

Synchronization primitive instructions are not supported in ARMv6-M. In order to implement atomic semaphores system software must provide the necessary access guarantees, for example by disabling interrupts or executing an appropriate kernel handler.

A3.7.2 Observability and completion

The concept of observability applies to all memory, however, the concept of global observability only applies to Shareable memory. Normal, Device and Strongly Ordered memory are defined in *Memory types* on page A3-8.

For all memory:

- A write to a location in memory is said to be observed by a memory system agent (the observer) when a subsequent read of the location by the observer returns the value written by the write.
- A write to a location in memory is said to be globally observed when a subsequent read of the location by any memory system agent returns the value written by the write.
- A read to a location in memory is said to be observed by a memory system agent (the observer) when a subsequent write of the location by the observer has no effect on the value returned by the read.
- A read to a location in memory is said to be globally observed when a subsequent write of the location by any memory system agent has no effect on the value returned by the read.

Additionally, for Strongly Ordered memory:

- A read or write to a memory mapped location in a peripheral which exhibits side-effects is said to be observed, and globally observed, only when the read or write meets the general conditions listed, can begin to affect the state of the memory-mapped peripheral, and can trigger any side effects that affect other peripheral devices, cores and/or memory.

For all memory, a read or write is defined to be complete when it is globally observed:

- A branch predictor maintenance operation is defined to be complete when the effects of operation are globally observed.

To determine when any side effects have completed, it is necessary to poll a location associated with the device, for example, a status register.

Side effect completion in Strongly Ordered and Device memory

For all memory-mapped peripherals, where the side-effects of a peripheral are required to be visible to the entire system, the peripheral must provide an IMPLEMENTATION DEFINED location which can be read to determine when all side effects are complete.

This is a key element of the architected memory order model.

A3.7.3 Ordering requirements for memory accesses

ARMv6-M defines access restrictions in the memory ordering allowed, depending on the memory attributes of the accesses involved. Table A3-7 on page A3-18 shows the memory ordering between two explicit accesses A1 and A2, where A1 (as listed in the first column) occurs before A2 (as listed in the first row) in program order.

The symbols used in Table A3-7 are as follows:

<	Accesses must be globally observed in program order, that is, A1 must be globally observed strictly before A2.
(blank)	Accesses can be globally observed in any order, provided that the requirements of uniprocessor semantics, for example respecting dependencies between instructions within a single processor, are maintained.

Table A3-7 Memory order restrictions

A2	Normal Access	Device Access		Strongly Ordered Access
A1		(Non-Shareable)	(Shareable)	
Normal Access				<
Device Access (Non-Shareable)		<		<
Device Access (Shareable)			<	<
Strongly Ordered Access	<	<	<	<

There are no ordering requirements for implicit accesses to any type of memory.

A3.7.4 Program order for instruction execution

Program order of instruction execution is the order of the instructions in the control flow trace. Explicit memory accesses in an execution can be either:

Strictly Ordered Denoted by <. Must occur strictly in order.

Ordered Denoted by <=. Must occur either in order, or simultaneously.

Multiple load and store instructions, such as `LDM{IA}`, `POP`, `STM{IA}`, and `PUSH` generate multiple word accesses, each of which is a separate access for the purpose of determining ordering.

The rules for determining program order for two accesses A1 and A2 are:

If A1 and A2 are generated by two different instructions:

- A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order
- A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

If A1 and A2 are generated by the same instruction:

- If A1 and A2 are two word loads generated by an `LDM` or `POP` instruction, or two word stores generated by an `STM` or `PUSH` instruction, excluding `LDM` or `POP` instructions whose register list includes the PC:
  - A1 <= A2 if the address of A1 is less than the address of A2
  - A2 <= A1 if the address of A2 is less than the address of A1.
- If A1 and A2 are two word loads generated by an `LDM` or `POP` instruction whose register list includes the PC, the program order of the memory operations is not defined.

A3.7.5 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, used to force synchronization events by a processor with respect to retiring load/store instructions. A memory barrier is used to guarantee completion of preceding load/store instructions to the programmers' model, flushing of any prefetched instructions prior to the event, or both. ARMv6-M includes three explicit barrier instructions to support the memory order model.

- Data Memory Barrier (DMB) as described in *Data Memory Barrier (DMB)*
- Data Synchronization Barrier (DSB) as described in *Data Synchronization Barrier (DSB)* on page A3-20
- Instruction Synchronization Barrier (ISB) as described in *Instruction Synchronization Barrier (ISB)* on page A3-20

Memory barriers affect explicit reads and writes to the memory system generated by load and store instructions being executed in the processor. Reads and writes generated by DMA transactions and instruction fetches are not explicit accesses.

Note

For information on barriers and correctness with respect to system configuration in the M-profile, see *Barrier support for system correctness* on page B2-9.

A3.7.6 Data Memory Barrier (DMB)

DMB acts as a data memory barrier, exhibiting the following behavior:

- All explicit memory accesses by instructions occurring in program order before this instruction are globally observed before any explicit memory accesses due to instructions occurring in program order after this instruction are observed.
- The `DMB` instruction has no effect on the ordering of other instructions executing on the processor.

As such, `DMB` ensures the apparent order of the explicit memory operations before and after the instruction, without ensuring their completion. For details on the `DMB` instruction, see *DMB* on page A6-41.

A3.7.7 Data Synchronization Barrier (DSB)

The DSB instruction operation acts as a special kind of Data Memory Barrier. The DSB operation completes when all explicit memory accesses before this instruction complete.

In addition, no instruction subsequent to the DSB can execute until the DSB completes. For details on the DSB instruction, see *DSB* on page A6-43.

A3.7.8 Instruction Synchronization Barrier (ISB)

The ISB instruction flushes the pipeline in the processor, so that all instructions following the pipeline flush are fetched from memory after the instruction has been completed. It ensures that the effects of context altering operations, such as branch predictor maintenance operations, as well as all changes to the special-purpose registers where applicable (see *The special-purpose control register* on page B1-10) executed before the ISB instruction are visible to the instructions fetched after the ISB.

In addition, the ISB instruction ensures that any branches which appear in program order after the ISB are always written into the branch prediction logic with the context that is visible after the ISB. This is required to ensure correct execution of the instruction stream.

Any context altering operations appearing in program order after the ISB only take effect after the ISB has been executed. This is due to the behavior of the context altering instructions.

ARM implementations are free to choose how far ahead of the current point of execution they prefetch instructions; either a fixed or a dynamically varying number of instructions. As well as being free to choose how many instructions to prefetch, an ARM implementation can choose which possible future execution path to prefetch along. For example, after a branch instruction, it can choose to prefetch either the instruction following the branch or the instruction at the branch target. This is known as branch prediction.

A potential problem with all forms of instruction prefetching is that the instruction in memory can be changed after it was prefetched but before it is executed. If this happens, the modification to the instruction in memory does not normally prevent the already prefetched copy of the instruction from executing to completion. The ISB and memory barrier instructions (DMB or DSB as appropriate) are used to force execution ordering where necessary.

For details on the ISB instruction, see *ISB* on page A6-46.

A3.8 Caches and memory hierarchy

Support for caches in ARMv6-M is limited to memory attributes. These can be exported on a supporting bus protocol such as AMBA (AHB or AXI protocols) to support system caches.

In situations where a breakdown in coherency can occur, software must manage the caches using cache maintenance operations which are memory mapped and IMPLEMENTATION DEFINED.

A3.8.1 Introduction to caches

A cache is a block of high-speed memory locations containing both address information (commonly known as a TAG) and the associated data. The purpose is to increase the average speed of a memory access. Caches operate on two principles of locality:

Spatial locality	an access to one location is likely to be followed by accesses from adjacent locations, for example sequential instruction execution or usage of a data structure
Temporal locality	an access to an area of memory is likely to be repeated within a short time period, for example execution of a code loop

To minimize the quantity of control information stored, the spatial locality property is used to group several locations together under the same TAG. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the processor wants to access a cacheable location, the cache is checked. If the access is a cache hit, the access occurs immediately, otherwise a location is allocated and the cache line loaded from memory. Different cache topologies and access policies are possible, however they must comply with the memory coherency model of the underlying architecture.

Caches introduce a number of potential problems, mainly because of:

- memory accesses occurring at times other than when the programmer would normally expect them
- the existence of multiple physical locations where a data item can be held.

A3.8.2 Implication of caches to the application programmer

Caches are largely invisible to the application programmer, but can become visible due to a breakdown in coherency. Such a breakdown can occur when:

- memory locations are updated by other agents in the systems
- memory updates made from the application code must be made visible to other agents in the system.

For example:

In systems with a DMA that reads memory locations which are held in the data cache of a processor, a breakdown of coherency occurs when the processor has written new data in the data cache, but the DMA reads the old data held in memory.

In a Harvard architecture of caches, a breakdown of coherency occurs when new instruction data has been written into the data cache and/or to memory, but the instruction cache still contains the old instruction data.

Chapter A4
The ARMv6-M Instruction Set

This chapter describes the Thumb instruction set as it applies to ARMv6-M. It contains the following sections:

A4.1 About the instruction set

ARMv6-M supports the Thumb instruction set including a small number of 32-bit instructions introduced with Thumb-2 technology, see *32-bit Thumb instruction encoding* on page A5-11. The 16-bit instruction support is equivalent to the Thumb instruction set support in ARMv6 prior to the introduction of Thumb-2 technology. This chapter describes the functionality available in the ARMv6-M Thumb instruction set, and the *Unified Assembler Language* (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers.
- A small number of 16-bit instructions can access the high registers, R8-R15.

The ARM and Thumb instruction sets are designed to *interwork* freely. Because ARMv6-M only supports Thumb instructions, interworking instructions in ARMv6-M must only reference Thumb state execution, see *ARMv6-M and interworking support* for more details.

In addition, see:

- Chapter A5 *Thumb Instruction Set Encoding* for encoding details of the Thumb instruction set
- Chapter A6 *Thumb Instruction Details* for detailed descriptions of the instructions.

A4.1.1 ARMv6-M and interworking support

Thumb interworking is held as bit [0] of an *interworking address*. Interworking addresses are used in the following instructions: `BX`, `BLX`, or `POP` that loads the PC.

ARMv6-M only supports the Thumb instruction execution state, therefore the value of address bit [0] must be 1 in interworking instructions, otherwise a fault occurs. All instructions ignore bit [0] and write bits [31:1]:'0' when updating the PC.

16-bit instructions that update the PC behave as follows:

- `ADD` (register) and `MOV` (register) branch within Thumb state without interworking

  Note

  The use of Rd as the PC in the `ADD` (SP plus register) 16-bit instruction is deprecated.

- `B`, or the `B<c>` instruction, branches without interworking
- `BLX` (register) and `BX` interwork on the value in Rm
- `POP` interworks on the value loaded to the PC
- `BKPT` and `SVC` cause exceptions and are not considered to be interworking instructions.

For more details, see the description of the `BXWritePC()` function in *Pseudocode details of ARM core register operations* on page A2-11.

The 32-bit `BL` instruction branches to Thumb state based on the instruction encoding, not due to bit [0] of the value written to the PC. It is the only 32-bit instruction in ARMv6-M that updates the PC.

A4.1.2 Conditional execution

Conditionally executed means that the instruction only has its normal effect on the programmers' model operation and memory if the N, Z, C and V flags in the APSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect.

Conditional execution in ARMv6-M can only be achieved using a 16-bit conditional branch instruction, with a branch range of –256 to +254 bytes. See *B* on page A6-27 for details.

See *Conditional execution* on page A6-8 for more information about conditional execution.

——— **Note** ———

The Thumb instruction set in other architecture variants supports additional conditional execution capabilities:

- a 32-bit conditional branch with a larger branch range
- a 16-bit Compare and Branch on Zero and a Compare and Branch on Nonzero instructions,
- an If-Then (`IT`) instruction.

These instructions are not supported in ARMv6-M.

————————

A4.2 Unified Assembler Language

This document uses the ARM *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all ARM and Thumb instructions. ARM Limited recommends the use of UAL for flexibility and maximum portability across all ARM architecture variants.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Note

Most earlier Thumb assembly language mnemonics are *not* supported. See Appendix B *Legacy Instruction Mnemonics* for details.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality.

Syntax options exist to ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

ARMv6-M only supports a single width of instruction for any given mnemonic. This makes the selection syntax valid but less relevant in the ARMv6-M case. The selection syntax might be useful for code sharing cases with other architecture variants.

A4.2.1 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC,4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a Thumb instruction, or plus 8 for an ARM instruction. The `Align(PC,4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC,4)` values for an ARM instruction, but there can be for a Thumb instruction.

2. Calculate the offset from the `PC` or `Align(PC,4)` value of the instruction to the address of the labelled instruction or literal data item.

3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC,4)` value and adds the calculated offset to form the required address.

   Note

   For instructions that encode a subtraction operation, if the instruction cannot encode the calculated offset, but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- `B` and `BL`. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch.
- The assembler syntax of the `LDR` instruction can specify the label of a literal data item that is to be loaded. The encoding of the instruction specifies a zero-extended immediate offset that is added to the `Align(PC,4)` value of the instruction to form the address of the data item.
- `ADR`. The assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is added to the `Align(PC,4)` value of the instruction to form the address of the data item.

A4.3 Branch instructions

Table A4-1 summarizes the branch instructions supported in the ARMv6-M Thumb instruction set.

Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-27	Branch to target address	+/–1 MB
BL on page A6-31	Call a subroutine	+/–16 MB
BLX (register) on page A6-32	Call a subroutine[a]	Any
BX on page A6-33	Branch to target address[a]	Any

a. In ARMv6-M, the interworking address must maintain Thumb execution state, otherwise a fault occurs.

A4.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- *Standard data-processing instructions*. This group perform basic data-processing operations, and share a common format with some variations.
- *Shift instructions* on page A4-8.
- *Multiply instructions* on page A4-9.
- *Packing and unpacking instructions* on page A4-9.
- *Miscellaneous data-processing instructions* on page A4-9.

A4.4.1 Standard data-processing instructions

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand Rm.

In addition to placing a result in the destination register, most of these instructions set the condition code flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table A4-2 summarizes the main data-processing instructions in the Thumb instruction set. The instructions are classified and described as applicable in two sections in Chapter A6 *Thumb Instruction Details*, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
`ADC`	Add with Carry	-
`ADD`	Add	Thumb instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant. ARMv6-M supports a more limited capability.
`ADR`	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant.
`AND`	Bitwise AND	-
`BIC`	Bitwise Bit Clear	-
`CMN`	Compare Negative	Sets flags. Like `ADD` but with no destination register.
`CMP`	Compare	Sets flags. Like `SUB` but with no destination register.
`EOR`	Bitwise Exclusive OR	-

Table A4-2 Standard data-processing instructions (continued)

Mnemonic	Instruction	Notes
`MOV`	Copies operand to destination	Has only one operand. Constant support is limited to loading an 8-bit immediate value in ARMv6-M. If the operand is a shifted register, the instruction is an `LSL`, `LSR`, `ASR`, or `ROR` instruction instead. See *Shift instructions* for details.
`MVN`	Bitwise NOT	Has only one operand. ARMv6-M does not support any immediate or shift options.
`ORR`	Bitwise OR	-
`RSB`	Reverse Subtract	Subtracts first operand from second operand. ARMv6-M only supports an immediate value of 0.
`SBC`	Subtract with Carry	-
`SUB`	Subtract	-
`TST`	Test	Sets flags. Like `AND` but with no destination register.

A4.4.2 Shift instructions

Table A4-3 lists the shift instructions in the Thumb instruction set.

Table A4-3 Shift instructions

Instruction[a]	See
Arithmetic Shift Right	*ASR (immediate)* on page A6-25
Arithmetic Shift Right	*ASR (register)* on page A6-26
Logical Shift Left	*LSL (immediate)* on page A6-63
Logical Shift Left	*LSL (register)* on page A6-64
Logical Shift Right	*LSR (immediate)* on page A6-65
Logical Shift Right	*LSR (register)* on page A6-66
Rotate Right	*ROR (register)* on page A6-85

a. Rotate Right with Extend (`RRX`) is not supported in ARMv6-M.

A4.4.3 Multiply instructions

The only multiply instruction supported in ARMv6-M performs a 32x32 multiply and generates a 32-bit result, see *MUL* on page A6-73. The instruction can operate on signed or unsigned quantities.

A4.4.4 Packing and unpacking instructions

Table A4-4 lists the packing and upacking instructions in the Thumb instruction set.

Table A4-4 Packing and unpacking instructions

Instruction	See	Operation
Signed Extend Byte	*SXTB* on page A6-104	Extend 8 bits to 32
Signed Extend Halfword	*SXTH* on page A6-105	Extend 16 bits to 32
Unsigned Extend Byte	*UXTB* on page A6-107	Extend 8 bits to 32
Unsigned Extend Halfword	*UXTH* on page A6-108	Extend 16 bits to 32

A4.4.5 Miscellaneous data-processing instructions

Table A4-5 lists the miscellaneous data-processing instructions in the Thumb instruction set. Immediate values in these instructions are simple binary numbers.

Table A4-5 Miscellaneous data-processing instructions

Instruction	See	Notes
Byte-Reverse Word	*REV* on page A6-82	-
Byte-Reverse Packed Halfword	*REV16* on page A6-83	-
Byte-Reverse Signed Halfword	*REVSH* on page A6-84	-

A4.5 Status register access instructions

The `MRS` and `MSR` instructions move the contents of the *Application Program Status Register* (APSR) to or from a general-purpose register.

The APSR is described in *The Application Program Status Register (APSR)* on page A2-12.

The condition flags in the APSR are normally set by executing data-processing instructions, and are normally used to control the execution of conditional instructions. However, you can set the flags explicitly using the `MSR` instruction, and you can read the current state of the flags explicitly using the `MRS` instruction.

For details of the system level use of status register access instructions, see Chapter B4 *ARMv6-M System Instructions*.

A4.6 Load and store instructions

Table A4-6 summarizes the general-purpose register load and store instructions in the Thumb instruction set. See also *Load/store multiple instructions* on page A4-13.

Load and store instructions have several options for addressing memory. See *Addressing modes* on page A4-12 for more information.

Table A4-6 Load and store instructions

Data type	Load	Store
32-bit word	LDR	STR
16-bit halfword	-	STRH
16-bit unsigned halfword	LDRH	-
16-bit signed halfword	LDRSH	-
8-bit byte	-	STRB
8-bit unsigned byte	LDRB	-
8-bit signed byte	LDRSB	-

A4.6.1 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

A4.6.2 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

In ARMv6-M, the base register is one of the R0-R7, SP or PC general-purpose registers.

For loads, the base register can be the PC. This permits PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter A6 *Thumb Instruction Details* are PC-relative loads.

In ARMv6-M, the address offset takes one of two formats:

Immediate	The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.
Register	The offset is a value from a general-purpose register. This register cannot be the PC. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

For more information on address mode support in ARMv6-M, see *Memory accesses* on page A6-12.

Note

Support for one or both formats and the range of permitted immediate values is instruction encoding dependent. See Chapter A6 *Thumb Instruction Details* for full details for each instruction.

A4.7 Load/store multiple instructions

Load Multiple instructions load a subset of the general-purpose registers from memory.

Store Multiple instructions store a subset of the general-purpose registers to memory.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and are either above or below the value in the base register. The base register can be updated by the total size of the data transferred. See the appropriate instruction behavior for exact details.

Table A4-7 summarizes the load/store multiple instructions in the ARMv6-M Thumb instruction set.

Table A4-7 Load/store multiple instructions

Instruction	Description
Load Multiple, Increment After or Full Descending	*LDM / LDMIA / LDMFD* on page A6-47
Pop multiple registers off the stack [a]	*POP* on page A6-79
Push multiple registers onto the stack [b]	*PUSH* on page A6-81
Store Multiple, Increment After or Empty Ascending	*STM / STMIA / STMEA* on page A6-89

a. This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.
b. This instruction decrements the base register before the memory access and updates the base register.

A4.7.1 Loads to the PC

The `POP` instruction can be used to load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode details of ARM core register operations* on page A2-11.

A4.8 Miscellaneous instructions

Table A4-8 summarizes the miscellaneous instructions in the ARMv6-M Thumb instruction set.

Table A4-8 Miscellaneous instructions

Instruction	See
Data Memory Barrier	*DMB* on page A6-41
Data Synchronization Barrier	*DSB* on page A6-43
Instruction Synchronization Barrier	*ISB* on page A6-46
No Operation	*NOP* on page A6-77
Send Event	*SEV* on page A6-88
Supervisor Call	*SVC (formerly SWI)* on page A6-103
Wait for Event	*WFE* on page A6-109
Wait for Interrupt	*WFI* on page A6-110
Yield	*YIELD* on page A6-111

A4.9 Exception-generating instructions

The following instructions are intended specifically to cause a processor exception to occur:

- The Supervisor Call (`SVC`, formerly `SWI`) instruction is used to cause an SVC exception to occur. This is the main mechanism in the ARM architecture for unprivileged (User) code to make calls to privileged Operating System code. See *Exception model* on page B1-13 for details.

  Note

  ARMv6-M execution is always privileged, however supervisor calls might be used to maintain a software hierarchy between application code and a system kernel.

- The Breakpoint (`BKPT`) instruction provides for software breakpoints. It can cause a running system to halt depending on the debug configuration. See *Debug event behavior* on page C1-10 for more details.

Chapter A5
Thumb Instruction Set Encoding

This chapter introduces the Thumb instruction set and describes how it uses the ARM programmers' model. It contains the following sections:

A5.1 Thumb instruction set encoding

The Thumb instruction stream is a sequence of halfword-aligned halfwords. Each Thumb instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If bits [15:11] of the halfword being decoded take any of the following values, the halfword is the first halfword of a 32-bit instruction:

- 0b11101
- 0b11110
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.

See *16-bit Thumb instruction encoding* on page A5-4 for details of the encoding of 16-bit Thumb instructions.

See *32-bit Thumb instruction encoding* on page A5-11 for details of the encoding of 32-bit Thumb instructions.

A5.1.1 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE.
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in this chapter.

An instruction is UNPREDICTABLE if:

- a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively, and the pseudocode for that encoding does not indicate that a different special case applies
- it is declared as UNPREDICTABLE in an instruction description or in this chapter.

Unless otherwise specified, Thumb instructions present in other architecture variants are UNDEFINED in ARMv6-M.

A5.1.2 Use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in Thumb instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings are:

- Read the PC value, that is, the address of the current instruction + 4. (Some instructions read the PC value implicitly, without the use of a register specifier, for example the conditional branch instruction `B<c>`.)
- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero. This allows PC-relative data addressing as used by the `ADR` and `LDR` (literal) instructions. The register specifier is implicit in the ARMv6-M encodings of these instructions.

For register writes, these meanings are:

- The PC can be specified as the destination register of an instruction. Thumb interworking defines whether bit [0] of the address is ignored or determines the instruction execution state. If it selects the execution state after the branch, bit [0] must have the value 1.

  Instructions can write the PC either implicitly (for example, `B<cond>`) or by using a register mask rather than a register specifier (`POP`). The address to branch to can be a loaded value (for example, `POP`), a register value (for example, `BX`), or the result of a calculation (for example, `ADD`).

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. This use does not apply to ARMv6-M encodings.

A5.1.3 Usage of 0b1101 as a register specifier

R13 is defined in Thumb such that its usage model is required to be that of a stack pointer, aligning R13 with the *ARM Architecture Procedure Call Standard* (AAPCS), the architecture usage model supported by the `PUSH` and `POP` instructions.

R13<1:0> definition

Bits [1:0] of R13 are treated as *SBZP* (Should Be Zero or Preserved). Writing a non-zero value to bits [1:0] results in UNPREDICTABLE behavior. Reading bits [1:0] returns zero.

R13 instruction support

R13 instruction support in ARMv6-M is restricted to the following:

- R13 as the source or destination register of a `MOV (register)` instruction:

```
MOV     SP,Rm
MOV     Rd,SP
```

- Adjusting R13 up or down by a multiple of its alignment:

```
SUB (SP minus immediate)
ADD (SP plus immediate)
ADD (SP plus register)     // where Rm is a multiple of 4
```

- R13 as the first operand `<Rm>` in an `ADD (SP plus register)` where Rd is not the SP.

- R13 as the first operand `<Rn>` in a `CMP (register)` instruction. `CMP` can be useful for stack checking.

- R13 as the address in a `POP` or `PUSH` instruction.

The restrictions affect:

- the high register form of `ADD` (register) and `CMP` (register), where the use of R13 as `<Rm>` is deprecated
- the `ADD (SP plus register)` where Rd == 13 and Rm is not word-aligned.

A5.2 16-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode															

Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-5
010000	*Data processing* on page A5-6
010001	*Special data instructions and branch and exchange* on page A5-7
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-51
0101xx	*Load/store single data item* on page A5-8
011xxx	
100xxx	
10100x	Generate PC-relative address, see *ADR* on page A6-23
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-21
1011xx	*Miscellaneous 16-bit instructions* on page A5-9
11000x	Store multiple registers, see *STM / STMIA / STMEA* on page A6-89
11001x	Load multiple registers, see *LDM / LDMIA / LDMFD* on page A6-47
1101xx	*Conditional branch, and supervisor call* on page A5-10
11100x	Unconditional Branch, see *B* on page A6-27

A5.2.1 Shift (immediate), add, subtract, move, and compare

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	opcode													

Table A5-2 shows the allocation of encodings in this space.

Table A5-2 16-bit Thumb encoding

opcode	Instruction	See
000xx	Logical Shift Left	*LSL (immediate)* on page A6-63
001xx	Logical Shift Right	*LSR (immediate)* on page A6-65
010xx	Arithmetic Shift Right	*ASR (immediate)* on page A6-25
01100	Add register	*ADD (register)* on page A6-19
01101	Subtract register	*SUB (register)* on page A6-101
01110	Add 3-bit immediate	*ADD (immediate)* on page A6-17
01111	Subtract 3-bit immediate	*SUB (immediate)* on page A6-99
100xx	Move	*MOV (immediate)* on page A6-67
101xx	Compare	*CMP (immediate)* on page A6-35
110xx	Add 8-bit immediate	*ADD (immediate)* on page A6-17
111xx	Subtract 8-bit immediate	*SUB (immediate)* on page A6-99

A5.2.2 Data processing

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	opcode									

Table A5-3 shows the allocation of encodings in this space.

Table A5-3 16-bit Thumb data processing instructions

opcode	Instruction	See
0000	Bitwise AND	*AND (register)* on page A6-24
0001	Exclusive OR	*EOR (register)* on page A6-45
0010	Logical Shift Left	*LSL (register)* on page A6-64
0011	Logical Shift Right	*LSR (register)* on page A6-66
0100	Arithmetic Shift Right	*ASR (register)* on page A6-26
0101	Add with Carry	*ADC (register)* on page A6-15
0110	Subtract with Carry	*SBC (register)* on page A6-87
0111	Rotate Right	*ROR (register)* on page A6-85
1000	Set flags on bitwise AND	*TST (register)* on page A6-106
1001	Reverse Subtract from 0	*RSB (immediate)* on page A6-86
1010	Compare Registers	*CMP (register)* on page A6-37
1011	Compare Negative	*CMN (register)* on page A6-34
1100	Logical OR	*ORR (register)* on page A6-78
1101	Multiply Two Registers	*MUL* on page A6-73
1110	Bit Clear	*BIC (register)* on page A6-29
1111	Bitwise NOT	*MVN (register)* on page A6-75

A5.2.3 Special data instructions and branch and exchange

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	opcode									

Table A5-4 shows the allocation of encodings in this space.

Table A5-4 Special data instructions and branch and exchange

opcode	Instruction	See
00xx	Add Registers	*ADD (register)* on page A6-19
0100	UNPREDICTABLE	
0101	Compare Registers	*CMP (register)* on page A6-37
011x		
10xx	Move Registers	*MOV (register)* on page A6-69
110x	Branch and Exchange	*BX* on page A6-33
111x	Branch with Link and Exchange	*BLX (register)* on page A6-32

A5.2.4 Load/store single data item

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opA				opB											

These instructions have one of the following values in opA:

- 0b0101
- 0b011x
- 0b100x.

Table A5-5 shows the allocation of encodings in this space.

Table A5-5 16-bit Thumb Load/store instructions

opA	opB	Instruction	See
0101	000	Store Register	*STR (register)* on page A6-93
0101	001	Store Register Halfword	*STRH (register)* on page A6-97
0101	010	Store Register Byte	*STRB (register)* on page A6-95
0101	011	Load Register Signed Byte	*LDRSB (register)* on page A6-61
0101	100	Load Register	*LDR (register)* on page A6-53
0101	101	Load Register Halfword	*LDRH (register)* on page A6-59
0101	110	Load Register Byte	*LDRB (register)* on page A6-56
0101	111	Load Register Signed Halfword	*LDRSH (register)* on page A6-62
0110	0xx	Store Register	*STR (immediate)* on page A6-91
0110	1xx	Load Register	*LDR (immediate)* on page A6-49
0111	0xx	Store Register Byte	*STRB (immediate)* on page A6-94
0111	1xx	Load Register Byte	*LDRB (immediate)* on page A6-55
1000	0xx	Store Register Halfword	*STRH (immediate)* on page A6-96
1000	1xx	Load Register Halfword	*LDRH (immediate)* on page A6-57
1001	0xx	Store Register SP relative	*STR (immediate)* on page A6-91
1001	1xx	Load Register SP relative	*LDR (immediate)* on page A6-49

A5.2.5 Miscellaneous 16-bit instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	opcode											

Table A5-6 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-6 Miscellaneous 16-bit instructions

opcode	Instruction	See
0110011	Change Processor State	*CPS* on page A6-39
00000xx	Add Immediate to SP	*ADD (SP plus immediate)* on page A6-21
00001xx	Subtract Immediate from SP	*SUB (SP minus immediate)* on page A6-102
001000x	Signed Extend Halfword	*SXTH* on page A6-105
001001x	Signed Extend Byte	*SXTB* on page A6-104
001010x	Unsigned Extend Halfword	*UXTH* on page A6-108
001011x	Unsigned Extend Byte	*UXTB* on page A6-107
010xxxx	Push Multiple Registers	*PUSH* on page A6-81
101000x	Byte-Reverse Word	*REV* on page A6-82
101001x	Byte-Reverse Packed Halfword	*REV16* on page A6-83
101011x	Byte-Reverse Signed Halfword	*REVSH* on page A6-84
110xxxx	Pop Multiple Registers	*POP* on page A6-79
1110xxx	Breakpoint	*BKPT* on page A6-30
1111xxx	If-Then, and hints	*Hint instructions* on page A5-10

Hint instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	opA				opB			

Table A5-7 shows the allocation of encodings in this space.

Other encodings in this space are unallocated hints. They execute as NOPs, but software must not use them.

Table A5-7 Hint instructions

opA	opB	Instruction	See
xxxx	not 0000	UNDEFINED[a]	
0000	0000	No Operation hint	*NOP* on page A6-77
0001	0000	Yield hint	*YIELD* on page A6-111
0010	0000	Wait for Event hint	*WFE* on page A6-109
0011	0000	Wait for Interrupt hint	*WFI* on page A6-110
0100	0000	Send Event hint	*SEV* on page A6-88

a. The If-Then (`IT`) instruction is not supported in ARMv6-M. The encoding space is UNDEFINED.

A5.2.6 Conditional branch, and supervisor call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	opcode											

Table A5-8 shows the allocation of encodings in this space.

Table A5-8 Branch and supervisor call instructions

opcode	Instruction	See
not 111x	Conditional branch	*B* on page A6-27
1110	Permanently UNDEFINED	
1111	Supervisor call	*SVC (formerly SWI)* on page A6-103

A5.3 32-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op1		op2											op															

op1 != 0b00. If op1 == 0b00, a 16-bit instruction is encoded, see *16-bit Thumb instruction encoding* on page A5-4.

Table A5-9 shows the allocation of ARMv6-M Thumb encodings in this space.

Table A5-9 32-bit Thumb encoding

op1	op2	op	Instruction class
10	xxxx xxx	1	*Branch and Miscellaneous control*

A5.3.1 Branch and Miscellaneous control

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	op1											1	op2														

Table A5-10 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-10 Branch and Miscellaneous control instructions

op2	op1	Instruction	See
0x0	011100x	Move to Special Register	*MSR (register)* on page A6-72
0x0	0111011	-	*Miscellaneous control instructions* on page A5-12
0x0	011111x	Move from Special Register	*MRS* on page A6-72
010	1111111	Permanently UNDEFINED	-
1x1	-	Branch with Link	*BL* on page A6-31

Miscellaneous control instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1					1	0		0					op							

Table A5-11 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED in ARMv6-M.

Table A5-11 Miscellaneous control instructions

op	Instruction	See
0100	Data Synchronization Barrier	*DSB* on page A6-43
0101	Data Memory Barrier	*DMB* on page A6-41
0110	Instruction Synchronization Barrier	*ISB* on page A6-46

Chapter A6
Thumb Instruction Details

This chapter describes Thumb instruction support in ARMv6-M. It contains the following sections:

A6.1 Format of instruction descriptions

The instruction descriptions in the alphabetical lists of instructions in *Alphabetical list of ARMv6-M Thumb instructions* on page A6-14 normally use the following format:

- instruction section title
- introduction to the instruction
- instruction encoding(s) with architecture information
- assembler syntax
- pseudocode describing how the instruction operates
- exception information
- notes (where applicable).

Each of these items is described in more detail in the following subsections.

A few instruction descriptions describe alternative mnemonics for other instructions and use an abbreviated and modified version of this format.

A6.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

Parenthesized text is also used to document the former mnemonic in some cases where a mnemonic has been replaced entirely by another mnemonic in the new assembler syntax.

A6.1.2 Introduction to the instruction

The instruction section title is followed by text that briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

A6.1.3 Instruction encodings

The *Encodings* subsection contains a list of one or more instruction encodings. For reference purposes, each Thumb instruction encoding is labelled, T1, T2, T3...

Each instruction encoding description consists of:

- Information about which architecture variants include the particular encoding of the instruction. Thumb instructions present since ARMv4T are labelled as *all versions of the Thumb ISA*, otherwise:
  - ARMv5T* means all variants of ARM Architecture version 5 that include Thumb instruction support.
  - ARMv6-M means a Thumb-only variant of the ARM architecture microcontroller profile that is compatible with ARMv6 Thumb support prior to the introduction of Thumb-2 technology.

— ARMv7-M means a Thumb-only variant of the ARM architecture microcontroller profile that provides enhanced performance and functionality with respect to ARMv6-M through Thumb-2 technology and additional system features such as fault handling support.

Note

This manual does not provide architecture variant information about non-M profile variants of ARMv6 and ARMv7. For such information, see the *ARM Architecture Reference Manual*.

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. In some cases, multiple syntaxes are given. The correct one to use is sometimes indicated by annotations to the syntax, such as *Inside IT block* and *Outside IT block*. In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  Note

  The `IT` instruction is not supported in ARMv6-M. This means any reference to `!InITBlock()` in a pseudocode statement in the instruction definitions always returns `TRUE`.

  There can be more than one syntax that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants and other operands to the instruction.

  The assembly syntax documented for the encoding is chosen to be the simplest one that ensures selection of that encoding for all operand combinations supported by that encoding.

  The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers may wish to use simpler syntaxes when they are suitable for the operand combination, in order to produce more readable disassembled code.

- An encoding diagram. This is half-width for 16-bit Thumb encodings and full-width for 32-bit Thumb encodings. The 32-bit Thumb encodings use a double vertical line between the two halfwords to act as a reminder that 32-bit Thumb encodings use the byte order of a sequence of two halfwords rather than of a word, as described in *Instruction alignment and byte ordering* on page A3-5.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the later *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix E *Pseudocode definition*.

A6.1.4 Assembler syntax

The *Assembly syntax* subsection describes the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- One or more syntax prototype lines written in a typewriter font, using the conventions described in *Assembler syntax prototype line conventions* on page A6-4. Each prototype line documents the mnemonic and (where appropriate) operand parts of a full line of assembler code. When there is more than one such line, each prototype line is annotated to indicate required results of the encoding-specific pseudocode. For each instruction encoding, this information can be used to determine whether any instructions matching that encoding are available when assembling that syntax, and if so, which ones.

- The line *where:* followed by descriptions of all of the variable or optional fields of the prototype syntax line.

  Some syntax fields are standardized across all or most instructions. These fields are described in *Standard assembler syntax fields* on page A6-7.

  By default, syntax fields that specify registers (such as `<Rd>`, `<Rn>`, or `<Rt>`) are permitted to be any of R0-R12 or LR in Thumb instructions. These require that the encoding-specific pseudocode should set the corresponding integer variable (such as `d`, `n`, or `t`) to the corresponding register number (0-12 for R0-R12, 14 for LR). This can normally be done by setting the corresponding bitfield in the instruction (named Rd, Rn, Rt...) to the binary encoding of that number. In the case of 16-bit Thumb encodings, this bitfield is normally of length 3 and so the encoding is only available when one of R0-R7 was specified in the assembler syntax. It is also common for such encodings to use a bitfield name such as Rdn. This indicates that the encoding is only available if `<Rd>` and `<Rn>` specify the same register, and that the register number of that register is encoded in the bitfield if they do.

  The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or document other differences from the default rules for such fields. Typical extensions are to allow the use of the SP and/or the PC (using register numbers 13 and 15 respectively).

Note

The pre-UAL Thumb assembler syntax is incompatible with UAL and is not documented in the instruction sections.

Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

`< >` Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence simply requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for a Thumb instruction contains an item `<Rn>` and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as add = `TRUE`. The assembler must only use encodings that produce that output.

`{ }`	Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text. Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.
spaces	Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.
`+/-`	This indicates an optional `+` or `-` sign. If neither is coded, `+` is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. The { and } characters need to be encoded in a few places as part of a variable item. When this happens, the description of the variable item indicates how they must be used.

A6.1.5 Pseudocode describing how the instruction operates

The *Operation* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix E *Pseudocode definition*.

A6.1.6 Exception information

The *Exceptions* subsection contains a list of the exceptional conditions that can be caused by execution of the instruction.

Processor exceptions are listed as follows:

- Resets and interrupts (including NMI, PendSV and SysTick) are not listed. They can occur before or after the execution of any instruction, and in some cases during the execution of an instruction, but they are not in general caused by the instruction concerned.
- HardFault exceptions are listed for all instructions that perform explicit data memory accesses.

  All instruction fetches can cause HardFault exceptions. These are not caused by execution of the instruction and so are not listed.
- HardFault exceptions can occur for the following reasons and are listed in the appropriate instructions:
  - Thumb interworking information that indicates a change of execution state
  - execution of a `BKPT` instruction where the Debug extension is not supported or enabled
  - ~~execution of an `SVC` instruction where the OS extension is not supported.~~

  HardFault exceptions also occur when pseudocode indicates that the instruction is UNDEFINED. These exceptions are not listed.
- The SVCall exception is listed for the `SVC` instruction.

Note

For a summary of the different types of HardFault exceptions see *Fault behavior* on page B1-29.

A6.1.7 Notes

Where appropriate, additional notes about the instruction appear under further subheadings.

A6.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

`<c>` Is an optional field. It specifies the condition under which the instruction is executed. If `<c>` is omitted, it defaults to *always* (`AL`). For details see *Conditional execution* on page A4-3.

Note

`B<c>` is the only conditional instruction supported in ARMv6-M. Instances of `<c>` shown in other instructions must be omitted or defined as `AL` and their corresponding pseudocode function ConditionPassed() in the operation section always returns `TRUE`.

`<q>` Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

`.N` Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

`.W` Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither `.W` nor `.N` is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description.

Note

ARMv6-M only supports either 16-bit or 32-bit encodings for a given instruction. The `.N` and `.W` are optional and produce assembler errors if incorrectly used.

A6.3 Conditional execution

In ARMv6-M, the `B<c>` instruction can be executed conditionally, based on the values of the APSR condition flags. The available conditions are listed in Table A6-1.

Note

The Thumb instruction set supports conditional execution of most instructions (if they are not `AL`) in a preceding `IT` instruction, however, the `IT` instruction is not supported in ARMv6-M. This means any reference to `!InITBlock()` in a pseudocode statement in the instruction definitions always returns `TRUE` and the `<c>` suffix must be omitted or `AL` in all instruction mnemonics with the exception of `B<c>`.

Table A6-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [a]	Condition flags
0000	`EQ`	Equal	Equal	Z == 1
0001	`NE`	Not equal	Not equal, or unordered	Z == 0
0010	`CS` [b]	Carry set	Greater than, equal, or unordered	C == 1
0011	`CC` [c]	Carry clear	Less than	C == 0
0100	`MI`	Minus, negative	Less than	N == 1
0101	`PL`	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	`VS`	Overflow	Unordered	V == 1
0111	`VC`	No overflow	Not unordered	V == 0
1000	`HI`	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	`LS`	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	`GE`	Signed greater than or equal	Greater than or equal	N == V
1011	`LT`	Signed less than	Less than, or unordered	N != V
1100	`GT`	Signed greater than	Greater than	Z == 0 and N == V
1101	`LE`	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (`AL`) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand. ARMv6-M does not include floating point support.
b. `HS` (unsigned higher or same) is a synonym for `CS`.
c. `LO` (unsigned lower) is a synonym for `CC`.
d. `AL` is an optional mnemonic extension for always, except in `IT` instructions.

A6.3.1 Pseudocode details of conditional execution

The `CurrentCond()` pseudocode function has prototype:

```
bits(4) CurrentCond()
```

and returns the 4-bit 'cond' field of the encoding for the Branch instruction (see *B* on page A6-27).

The `ConditionPassed()` function uses this condition specifier and the APSR condition flags to determine whether the instruction must be executed:

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    cond = CurrentCond();

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                        // EQ or NE
        when '001'  result = (APSR.C == '1');                        // CS or CC
        when '010'  result = (APSR.N == '1');                        // MI or PL
        when '011'  result = (APSR.V == '1');                        // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');     // HI or LS
        when '101'  result = (APSR.N == APSR.V);                     // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0');  // GT or LE
        when '111'  result = TRUE;                                   // AL

    // Condition bits in the set "111x" indicate the instruction is always executed.
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

A6.4 Shifts applied to a register

Shifts only apply to the ASR, LSL, LSR, and ROR data-processing instructions in ARMv6-M. Other instructions are declared with shift type SRType_LSL and a shift value of zero where shift operations are supported by additional encodings in other architecture variants.

A6.4.1 Shift operations

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;


// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);
```

```
if amount == 0 then
    (result, carry_out) = (value, carry_in);
else
    case type of
        when SRType_LSL
            (result, carry_out) = LSL_C(value, amount);
        when SRType_LSR
            (result, carry_out) = LSR_C(value, amount);
        when SRType_ASR
            (result, carry_out) = ASR_C(value, amount);
        when SRType_ROR
            (result, carry_out) = ROR_C(value, amount);
        when SRType_RRX
            (result, carry_out) = RRX_C(value, carry_in);

return (result, carry_out);
```

A6.5 Memory accesses

The following addressing mode is commonly permitted in ARMv6-M for memory access instructions:

Offset addressing

The offset value is added to or subtracted from an address obtained from the base register. The result is used as the address for the memory access. The base register is unaltered.

The assembly language syntax for this mode is:

```
[<Rn>,<offset>]
```

`<Rn>` is the base register and `<offset>` can be:

- an immediate constant, such as `<imm3>` or `<imm8>`
- an index register, `<Rm>`.

For information about unaligned access and endianness, see:

- *Alignment support* on page A3-3
- *Endian support* on page A3-4.

ARMv6-M does not support exclusive access to memory, see *Synchronization and semaphores* on page A3-7.

A6.6 Hint Instructions

Two classes of hint instruction exist within the Thumb instruction set:

- memory hints
- NOP-compatible hints.

Only 16-bit versions of the NOP-compatible hints are supported in ARMv6-M. For information on the 16-bit encodings see *Hint instructions* on page A5-10.

A6.7 Alphabetical list of ARMv6-M Thumb instructions

Every ARMv6-M Thumb instruction is listed in this section. See *Format of instruction descriptions* on page A6-2 for details of the format used.

A6.7.1 ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ADCS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`ADCS<c><q>  {<Rd>,} <Rn>, <Rm>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is optionally shifted and used as the second operand.

The pre-UAL syntax `ADC<c>S` is equivalent to `ADCS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.2 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

```
ADDS <Rd>,<Rn>,#<imm3>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

```
ADDS <Rdn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Assembler syntax

`ADDS<c><q>  {<Rd>,} <Rn>, #<const>`	All encodings permitted

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand. If the SP is specified for `<Rn>`, see *ADD (SP plus immediate)* on page A6-21. If the PC is specified for `<Rn>`, see *ADR* on page A6-23.
`<const>`	The immediate value to be added to the value obtained from `<Rn>`. The range of allowed values is 0-7 for encoding T1, and 0-255 for encoding T2. Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.3 ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. Encoding T1 updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ADDS <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb ISA.

`ADD<c> <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DN	Rm				Rdn		

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`S`	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>` and encoding T2 is preferred to encoding T1 if both are available. If `<Rd>` is specified, encoding T1 is preferred to encoding T2.
`<Rn>`	The register that contains the first operand. If the SP is specified for `<Rn>`, see *ADD (SP plus register)* on page A6-22.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A6.7.4 ADD (SP plus immediate)

This instruction adds an immediate value to the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

`ADD<c> <Rd>,SP,#<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

Encoding T2 All versions of the Thumb ISA.

`ADD<c> SP,SP,#<imm7>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Assembler syntax

```
ADD<c><q>  {<Rd>,} SP, #<const>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is SP.
`<const>`	The immediate value to be added to the value obtained from `<Rn>`. Allowed values are multiples of 4 in the range 0-1020 for encoding T1and multiples of 4 in the range 0-508 for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    R[d] = result;

    // no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.5 ADD (SP plus register)

This instruction adds a register value to the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

`ADD<c> <Rdm>, SP, <Rdm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DM	1	1	0	1	Rdm		

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb ISA.

`ADD<c> SP,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	Rm				1	0	1

```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
ADD<c><q>  {<Rd>,} SP, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is SP.
`<Rm>`	The register that is used as the second operand. This register can be the SP, but such instructions are deprecated and the instruction can only be `ADD SP,SP,SP`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, ‘0’);
    if d == 15 then
        ALUWritePC(result);
    else
        R[d] = result;    // no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.6 ADR

Address to Register adds an immediate value to the PC value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

```
ADR<c> <Rd>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Assembler syntax

`ADR<c><q>  <Rd>, <label>`	Normal syntax
`ADD<c><q>  <Rd>, PC, #<const>`	Alternative syntax

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<label>`	The label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label. Only a positive value is permitted with `imm32` equal to the offset. Allowed values of the offset are multiples of four in the range 0 to 1020 for encoding T1.

In the alternative syntax forms:

`<const>`	The offset value for the `ADD` form. Allowed values are multiples of four in the range 0 to 1020 for encoding T1.

Note

It is recommended that the alternative syntax form is avoided where possible.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[d] = result;
```

Exceptions

None.

A6.7.7 AND (register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ANDS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
0	1	0	0	0	0	0	0	0	0	Rm	Rdn

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`ANDS<c><q>  {<Rd>,} <Rn>, <Rm>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `AND<c>S` is equivalent to `ANDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.8 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ASRS <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('10', imm5);
```

Assembler syntax

`ASRS<c><q>  <Rd>, <Rm>, #<imm5>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the first operand.
`<imm5>`	The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.9 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

```
ASRS   <Rdn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Assembler syntax

```
ASRS<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.10 B

Branch causes a branch to a target address.

Encoding T1 All versions of the Thumb ISA.

`B<c> <label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb ISA.

`B<c> <label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
B<c><q>  <label>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Note

Encoding T1 is conditional.

For encodings T1, <c> is not allowed to be `AL` or omitted.

For ARMv6-M, <c> must be `AL` or omitted with encoding T2.

`<label>` The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that will set `imm32` to that offset.

Allowed offsets are even numbers in the range -256 to 254 for encoding T1 and -2048 to 2046 for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

A6.7.11 BIC (register)

Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`BICS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`BICS<c><q> {<Rd>,} <Rn>, <Rm>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `BIC<c>S` is equivalent to `BICS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.12 BKPT

Breakpoint causes a HardFault exception or a debug halt to occur depending on the presence and configuration of the debug support.

Encoding T1 ARMv5T*, ARMv6-M, ARMv7-M M profile specific behavior

```
BKPT #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

Assembler syntax

```
BKPT<q>  #<imm8>
```

where:

`<q>`	See *Standard assembler syntax fields* on page A6-7.
`<imm8>`	Specifies an 8-bit value that is stored in the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Operation

```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

Exceptions

HardFault.

A6.7.13 BL

Branch with Link (immediate) calls a subroutine at a PC-relative address.

Encoding T1 All versions of the Thumb ISA.

`BL<c> <label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
toARM = FALSE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

`BL<c><q>  <label>`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<label>`	The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding that will set `imm32` to that offset. Allowed offsets are even numbers in the range -16777216 to 16777214.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(PC + imm32);
```

Exceptions

None.

Note

Before the introduction of Thumb-2 technology, J1 and J2 in encodings T1 and T2 were both 1, resulting in a smaller branch range. The instructions could be executed as two separate 16-bit instructions, with the first instruction `instr1` setting LR to `PC + SignExtend(instr1<10:0>:'00000000000', 32)` and the second instruction completing the operation. It is no longer possible to split the `BL` instruction into two 16-bit instructions in ARMv6T2, ARMv6-M and ARMv7.

A6.7.14 BLX (register)

Branch with Link and Exchange calls a subroutine at an address and instruction set specified by a register. ARMv6-M only supports Thumb execution. An attempt to change the instruction execution state causes an exception.

Encoding T1 ARMv5T*, ARMv6-M, ARMv7-M

`BLX<c> <Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
BLX<c><q>  <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rm>`	The register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BLXWritePC(target);
```

Exceptions

HardFault.

A6.7.15 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register. ARMv6-M only supports Thumb execution. An attempt to change the instruction execution state causes an exception.

Encoding T1

All versions of the Thumb ISA.

```
BX<c> <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if m==15 then UNPREDICTABLE;
```

Assembler syntax

```
BX<c><q>  <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rm>`	The register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

Exceptions

HardFault.

A6.7.16 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`CMN<c> <Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
CMN<c><q>  <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.17 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`CMP<c> <Rn>,#<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							

```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

Assembler syntax

```
CMP<c><q>  <Rn>, #<const>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The register that contains the operand.
`<const>`	The immediate value to be added to the value obtained from `<Rn>`. The range of allowed values is 0-255 for encoding T1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.18 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb ISA.

`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` not both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm				Rn		

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

Assembler syntax

```
CMP<c><q>  <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The register that contains the first operand. The SP can be used.
`<Rm>`	The register that is optionally shifted and used as the second operand. The SP can be used, but use of the SP is deprecated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.19 CPS

Change Processor State. The instruction modifies the PRIMASK special-purpose register values.

Encoding T1

`CPS<effect> <iflags>` ARMv6-M enhanced functionality in ARMv7-M

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	(0)

Note

`CPS` is a system level instruction with ARMv6-M specific behavior. For the complete instruction definition see *CPS* on page B4-3.

A6.7.20 CPY

Copy is a pre-UAL synonym for `MOV` (register).

Assembler syntax

```
CPY  <Rd>, <Rn>
```

This is equivalent to:

```
MOV  <Rd>, <Rn>
```

Exceptions

None.

A6.7.21 DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. It does not affect the ordering of any other instructions executing on the processor.

Encoding T1 ARMv6-M, ARMv7-M

`DMB<c> #<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

```
// No additional decoding required
```

Assembler syntax

```
DMB<c><q>  {<opt>}
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<opt>`	Specifies an optional limitation on the DMB operation:

`SY`	DMB operation ensures ordering of all accesses, encoded as option == '1111'. Can be omitted.

All other encodings of the option are reserved. The corresponding instructions execute as system (SY) DMB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

Exceptions

None.

A6.7.22 DSB

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All system maintenance operations before this instruction complete.

Encoding T1 ARMv6-M, ARMv7-M

`DSB<c> #<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	option			

```
// No additional decoding required
```

Assembler syntax

```
DSB<c><q>  {<opt>}
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<opt>`	Specifies an optional limitation on the DSB operation:

`SY`	DSB operation ensures completion of all accesses, encoded as option == '1111'. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as system (SY) DSB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.23 EOR (register)

Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`EORS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
EORS<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `EOR<c>S` is equivalent to `EORS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.24 ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`. See *Barrier support for system correctness* on page B2-9 for more details.

In addition, the `ISB` instruction ensures that any branches that appear in program order after it are always written into any branch prediction logic with the context that is visible after the `ISB` instruction. This is required to ensure correct execution of the instruction stream.

Encoding T1 ARMv6-M, ARMv7-M

`ISB<c> #<option>`

15 14 13 12 11	10	9 8 7	6 5	4	3 2 1 0	15 14	13	12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 0	0	1 1 1	0 1	1	(1) (1) (1) (1)	1 0	(0)	0	(1) (1) (1) (1)	0 1 1 0	option

```
// No additional decoding required
```

Assembler syntax

`ISB<c><q>  {<opt>}`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<opt>`	Specifies an optional limitation on the ISB operation. Allowed values are:

`SY`	Full system ISB operation, encoded as option == '1111'. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system ISB operations, but should not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.25 LDM / LDMIA / LDMFD

Load Multiple Increment After loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations is written back to the base register when the base register is not part of the register list.

Encoding T1 All versions of the Thumb ISA.

`LDM<c> <Rn>!,<registers>`	`<Rn>` not included in `<registers>`
`LDM<c> <Rn>,<registers>`	`<Rn>` included in `<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

`LDM<c><q>  <Rn>{!}, <registers>`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The base register.
`!`	Causes the instruction to write a modified value back to `<Rn>`. If `!` is omitted, the instruction does not change `<Rn>` in this way.
`<registers>`	Is a list of one or more registers to be loaded, separated by commas and surrounded by `{` and `}`. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

`LDMIA` and `LDMFD` are pseudo-instructions for `LDM`. `LDMFD` refers to its use for popping data from Full Descending stacks.

The pre-UAL syntaxes `LDM<c>IA` and `LDM<c>FD` are equivalent to `LDM<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 7
        if registers<i> == ‘1’ then
            R[i] = MemA[address,4];  address = address + 4;

    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.26 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb ISA.

```
LDR<c> <Rt>,[SP{,#<imm8>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`LDR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]` Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1 and multiples of 4 in the range 0-1020 for encoding T2. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemU[address,4];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.27 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			imm8							

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Assembler syntax

`LDR<c><q>  <Rt>, <label>`	Normal syntax
`LDR<c><q>  <Rt>, [PC, #<imm>]`	Alternative syntax

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are multiples of four in the range 0 to 1020 for encoding T1.

In the alternative syntax form:

`<imm>`	The immediate offset added to the `Align(PC, 4)` value of the instruction to form the address. Allowed values are multiples of four in the range 0 to 1020 for encoding T1.

Note

It is recommended that the alternative syntax form is avoided where possible.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = MemU[address,4];
```

Exceptions

HardFault.

A6.7.28 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as an address or exception return value and a branch occurs. Bit [0] complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit [0] is 0, a HardFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
LDR<c><q>  <Rt>, [<Rn>, <Rm>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = MemU[address,4];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.29 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

`LDRB<c> <Rt>,[<Rn>{,#<imm5>}]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`LDRB<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]` Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-31 for encoding T1. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.30 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

`LDRB<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`LDRB<c><q>  <Rt>, [<Rn>, <Rm>]`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
```

Exceptions

HardFault.

A6.7.31 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

```
LDRH<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`LDRH<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to the value of `<Rn>` to form the address. Allowed values are multiples of 2 in the range 0-62 for encoding T1. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

HardFault.

A6.7.32 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

```
LDRH<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`LDRH<c><q>  <Rt>, [<Rn>, <Rm>]`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

HardFault.

A6.7.33 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

`LDRSB<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
LDRSB<c><q>  <Rt>, [<Rn>, <Rm>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
```

Exceptions

HardFault.

A6.7.34 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-12 for more information.

Encoding T1 All versions of the Thumb ISA.

`LDRSH<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`LDRSH<c><q>  <Rt>, [<Rn>, <Rm>]`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The destination register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Exceptions

HardFault.

A6.7.35 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSLS <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm			Rd		

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

Assembler syntax

`LSLS<c><q>  <Rd>, <Rm>, #<imm5>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the first operand.
`<imm5>`	The shift amount, in the range 0 to 31. See *Shifts applied to a register* on page A6-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.36 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSLS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Assembler syntax

```
LSLS<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.37 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSRS <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

Assembler syntax

`LSRS<c><q>  <Rd>, <Rm>, #<imm5>`

where:

`S`	The instruction updates the flags..
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the first operand.
`<imm5>`	The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.38 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSRS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Assembler syntax

```
LSRS<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.39 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

```
MOVS <Rd>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Assembler syntax

`MOVS<c><q>  <Rd>, #<const>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<const>`	The immediate value to be placed in `<Rd>`. The range of allowed values is 0-255 for encoding T1.

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.40 MOV (register)

Move (register) copies a value from a register to the destination register. Encoding T2 updates the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M If `<Rd>` and `<Rm>` both from R0-R7, otherwise all versions of the Thumb ISA.

`MOV<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
```

Encoding T2 All versions of the Thumb ISA.

`MOVS <Rd>,<Rm>` (formerly `LSL <Rd>,<Rm>,#0`)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
```

Assembler syntax

```
MOV{S}<c><q>  <Rd>, <Rm>
```

where:

`S`	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. This register can be the SP or PC, provided S is not specified. If <Rd> is the PC, the instruction causes a branch to the address moved to the PC.
`<Rm>`	The source register. This register can be the SP or PC, provided S is not specified.

Note

The use of `MOV` (register) instructions in which both <Rd> and <Rm> are the SP or PC is deprecated.

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            ~~APSR.C = carry;~~
            // APSR.V unchanged
```

Exceptions

None.

A6.7.41 MOV (shifted register)

Move (shifted register) is a pseudo-instruction for `ASR`, `LSL`, `LSR`, `ROR`, and `RRX`.

See the following sections for details:

- *ASR (immediate)* on page A6-25
- *ASR (register)* on page A6-26
- *LSL (immediate)* on page A6-63
- *LSL (register)* on page A6-64
- *LSR (immediate)* on page A6-65
- *LSR (register)* on page A6-66
- *ROR (register)* on page A6-85

Assembler syntax

Table A6-2 shows the equivalences between `MOV` (shifted register) and other instructions.

Table A6-2 MOV (shift, register shift) equivalences)

MOV instruction	Canonical form
`MOVS <Rd>,<Rm>,ASR #<n>`	`ASRS <Rd>,<Rm>,#<n>`
`MOVS <Rd>,<Rm>,LSL #<n>`	`LSLS <Rd>,<Rm>,#<n>`
`MOVS <Rd>,<Rm>,LSR #<n>`	`LSRS <Rd>,<Rm>,#<n>`
`MOVS <Rd>,<Rm>,ASR <Rs>`	`ASRS <Rd>,<Rm>,<Rs>`
`MOVS <Rd>,<Rm>,LSL <Rs>`	`LSLS <Rd>,<Rm>,<Rs>`
`MOVS <Rd>,<Rm>,LSR <Rs>`	`LSRS <Rd>,<Rm>,<Rs>`
`MOVS <Rd>,<Rm>,ROR <Rs>`	`RORS <Rd>,<Rm>,<Rs>`

The canonical form of the instruction is produced on disassembly.

Exceptions

None.

A6.7.42 MRS

Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose ARM register.

Encoding T1 ARMv6-M enhanced functionality in ARMv7-M

`MRS<c> <Rd>,<spec_reg>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				SYSm							

Note

`MRS` is a system level instruction except when accessing the APSR (SYSm = 0) or CONTROL register (SYSm = 0x14). For the complete instruction definition see *MRS* on page B4-5.

A6.7.43 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose ARM register to the specified special-purpose register.

Encoding T1 ARMv6-M enhanced functionality in ARMv7-M

`MSR<c> <spec_reg>,<Rn>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn				1	0	(0)	0	(1)	(0)	(0)	(0)	SYSm							

Note

`MSR`(register) is a system level instruction except when accessing the APSR (SYSm = 0). For the complete instruction definition see *MSR (register)* on page B4-8.

A6.7.44 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

```
MULS <Rdm>,<Rn>,<Rdm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

Assembler syntax

```
MULS<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.

Note

For ARMv6-M, <Rd> can only be omitted when d == n ==m. See *Assembler syntax prototype line conventions* on page A6-4 for the rule on optional arguments.

`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A6.7.45 MVN (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`MVNS <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`MVNS<c><q>  <Rd>, <Rm>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that is used as the source register.

The pre-UAL syntax `MVN<c>S` is equivalent to `MVNS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.46 NEG

Negate is a pre-UAL synonym for `RSB` (immediate) with an immediate value of 0. See *RSB (immediate)* on page A6-86 for details.

Assembler syntax

```
NEG<c><q>  {<Rd>,} <Rm>
```

This is equivalent to:

```
RSBS<c><q>  {<Rd>,} <Rm>, #0
```

Exceptions

None.

A6.7.47 NOP

No Operation does nothing. This instruction can be used for code alignment purposes.

This is a NOP-compatible hint (the architected NOP), see *Hint Instructions* on page A6-13.

See *Pre-UAL pseudo-instruction NOP* on page AppxB-6 for details of `NOP` before the introduction of UAL.

Note

The timing effects of including a `NOP` instruction in code are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. `NOP` instructions are therefore not suitable for timing loops.

Encoding T1

ARMv6-M, ARMv7-M

`NOP<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

```
NOP<c><q>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

Exceptions

None.

A6.7.48 ORR (register)

Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`ORRS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`ORRS<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `ORR<c>S` is equivalent to `ORRS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.49 POP

Pop Multiple Registers loads a subset (or possibly all) of the general-purpose registers R0-R7 and the PC from the stack.

If the registers loaded include the PC, the word loaded for the PC is treated as an address or an exception return value and a branch occurs. Bit [0] complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit [0] is 0, a HardFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
POP<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

```
registers = P:'0000000':register_list;  if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
POP<c><q>  <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;

    for i = 0 to 7
        if registers<i> == '1' then
            R[i} = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    SP = SP + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.50 PUSH

Push Multiple Registers stores a subset (or possibly all) of the general-purpose registers R0-R7 and the LR to the stack.

Encoding T1 All versions of the Thumb ISA.

`PUSH<c> <registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							

```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
PUSH<c><q>  <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored. The registers are stored in sequence, the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    SP = SP - 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.51 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

Encoding T1 ARMv6-M, ARMv7-M

`REV<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Assembler syntax

`REV<c><q>  <Rd>, <Rm>`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Exceptions

None.

A6.7.52 REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

Encoding T1 ARMv6-M, ARMv7-M

`REV16<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Assembler syntax

```
REV16<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

A6.7.53 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32 bits.

Encoding T1 ARMv6-M, ARMv7-M

`REVSH<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Assembler syntax

```
REVSH<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

A6.7.54 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`RORS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Assembler syntax

```
RORS<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register whose bottom byte contains the amount to rotate by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.55 RSB (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`RSBS <Rd>,<Rn>,#0`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

Assembler syntax

```
RSBS<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<const>`	The immediate value to be added to the value obtained from `<Rn>`. ARMv6-M only supports a value of 0.

The pre-UAL syntax `RSB<c>S` is equivalent to `RSBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.56 SBC (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`SBCS <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
SBCS<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `SBC<c>S` is equivalent to `SBCS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.57 SEV

Send Event is a hint instruction. It causes an event to be signaled to all CPUs within a multiprocessor system.

This is a NOP-compatible hint, see *Hint Instructions* on page A6-13.

Encoding T1 ARMv6-M, ARMv7-M

`SEV<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

`SEV<c><q>`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_SendEvent();
```

Exceptions

None.

A6.7.58 STM / STMIA / STMEA

Store Multiple Increment After (Store Multiple Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations is written back to the base register.

Encoding T1 All versions of the Thumb ISA.

```
STM<c> <Rn>!,<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
STM{IA|EA}<c><q>  <Rn>!, <registers>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The base register.
`!`	Causes the instruction to write a modified value back to `<Rn>`.
`<registers>`	Is a list of one or more registers to be stored, separated by commas and surrounded by `{` and `}`. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. If the base register is included and not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. Use of <Rn> in the register list is deprecated.

`STMEA` and `STMIA` are pseudo-instructions for `STM`, `STMEA` referring to its use for pushing data onto Empty Ascending stacks.

The pre-UAL syntaxes `STM<c>IA` and `STM<c>EA` are equivalent to `STM<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;    // encoding T1 only
            else
                MemA[address,4] = R[i];
            address = address + 4;

    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.59 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb ISA.

```
STR<c> <Rt>,[SP,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`STR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]` Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1 and multiples of 4 in the range 0-1020 for encoding T2. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.60 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1

All versions of the Thumb ISA.

`STR<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
STR<c><q>  <Rt>, [<Rn>, <Rm>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The register that contains the base value.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    data = R[t];
    MemU[address,4] = data;
```

Exceptions

HardFault.

A6.7.61 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

`STRB<c> <Rt>,[<Rn>,#<imm5>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`STRB<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]` Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-31 for encoding T1. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.62 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

`STRB<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
STRB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The base register.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,1] = R[t]<7:0>;
```

Exceptions

HardFault.

A6.7.63 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

`STRH<c> <Rt>,[<Rn>{,#<imm5>}]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

`STRH<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]` Offset: `index==TRUE`, `wback==FALSE`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The base register.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`).
`<imm>`	The immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 2 in the range 0-62 for encoding T1. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;

    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.64 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. See *Memory accesses* on page A6-12 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STRH<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
STRH<c><q>  <Rt>, [<Rn>, <Rm>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rt>`	The source register.
`<Rn>`	The base register.
`<Rm>`	Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,2] = R[t]<15:0>;
```

Exceptions

HardFault.

A6.7.65 SUB (immediate)

This instruction subtracts an immediate value from a register value, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb ISA.

`SUBS <Rd>,<Rn>,#<imm3>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

`SUBS <Rdn>,#<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Assembler syntax

```
SUBS<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<const>`	The immediate value to be subtracted from the value obtained from `<Rn>`. The range of allowed values is 0-7 for encoding T1 and 0-255 for encoding T2. Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.66 SUB (register)

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`SUBS <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

`SUBS<c><q>  {<Rd>,} <Rn>, <Rm>`

where:

`S`	The instruction updates the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.67 SUB (SP minus immediate)

This instruction subtracts an immediate value from the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

`SUB<c> SP,SP,#<imm7>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Assembler syntax

```
SUB<c><q>  {<Rd>,} SP, #<const>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register. If `<Rd>` is omitted, this register is SP.
`<const>`	The immediate value to be added to the value obtained from SP. Allowed values are multiples of 4 in the range 0-508 for encoding T1.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), ‘1’);
    R[d] = result;

    // no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.68 SVC (formerly SWI)

Generates an SVCall exception ~~when the ARMv6-M OS Extension is implemented,~~ see *Exceptions* on page B1-4 for more information. When the exception is escalated ~~or the OS Extension is not implemented,~~ a HardFault exception is caused.

Note

~~Where the OS Extension is not implemented, the instruction is UNDEFINED. For the impact on `ReturnAddress()` when SVC is UNDEFINED rather than escalated see *Exception entry behavior* on page B1-18.~~

Use it as a call to an operating system to provide a service.

Encoding T1 All versions of the Thumb ISA M profile specific behavior

`SVC<c> #<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Assembler syntax

```
SVC<c><q>  #<imm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<imm>`	Specifies an 8-bit immediate constant.

The pre-UAL syntax `SWI<c>` is equivalent to `SVC<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

Exceptions

SVCall, HardFault.

A6.7.69 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register..

Encoding T1 ARMv6-M, ARMv7-M

```
SXTB<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Assembler syntax

```
SXTB<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Exceptions

None.

A6.7.70 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6-M, ARMv7-M

`SXTH<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Assembler syntax

```
SXTH<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Exceptions

None.

A6.7.71 TST (register)

Test (register) performs a logical AND operation on two register values. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`TST<c> <Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	Rm			Rn		

```
n = UInt(Rdn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

```
TST<c><q>  <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	The register that contains the first operand.
`<Rm>`	The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.72 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6-M, ARMv7-M

`UXTB<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Assembler syntax

```
UXTB<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Exceptions

None.

A6.7.73 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6-M, ARMv7-M

`UXTH<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Assembler syntax

```
UXTH<c><q>  <Rd>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Exceptions

None.

A6.7.74 WFE

Wait For Event is a hint instruction that permits the processor to enter a low-power state until one of a number of events occurs, including events signaled by the `SEV` instruction on any processor in a multiprocessor system. For more information, see *Wait For Event and Send Event* on page B1-33.

For general hint behavior, see *Hint Instructions* on page A6-13.

Encoding T1 ARMv6-M, ARMv7-M

`WFE<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

`WFE<c><q>`

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

Exceptions

None.

A6.7.75 WFI

Wait For Interrupt is a hint instruction that suspends execution until one of a number of events occurs. For more information, see *Wait For Interrupt* on page B1-35.

For general hint behavior, see *Hint Instructions* on page A6-13.

Encoding T1 ARMv6-M, ARMv7-M

`WFI<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

`WFI<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

Exceptions

None.

Notes

PRIMASK If PRIMASK is set, an asynchronous exception that has a higher group priority than any active exception results in a `WFI` instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.

A6.7.76 YIELD

`YIELD` is a hint instruction. It allows software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spinlock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

For general hint behavior, see *Hint Instructions* on page A6-13.

Encoding T1 ARMv6-M, ARMv7-M

`YIELD<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

`YIELD<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Exceptions

None.

Part B

System Level Architecture

Chapter B1
System Level Programmers' Model

This chapter contains information on the system programmers' model. It describes the registers, exception model and fault handling capabilities. This chapter contains the following sections:

B1.1 Introduction to the system level

The ARM architecture is defined in a hierarchical manner, where the features are described in Chapter A2 *Application Level Programmers' Model* at the application level, with underlying system support. What features are available and how they are supported is defined in the architecture profiles, making the system level support profile specific.

ARMv6-M supports privileged execution only, always allowing configuration and control of the system resources. System support is typically provided by an operating system, which provides system services to the applications, either transparently, or through application initiated service calls. The operating system is also responsible for servicing interrupts and other system events, making exceptions a key component of the system level programmers' model. A consequence of privileged execution only is that application code can raise a supervisor call using `SVC` where it is supported (see below) or handle system access and control directly.

ARMv6-M is a subset of ARMv7-M. The subset breaks down into a mandatory set of features and ~~two~~ architecture extension~~s~~:

- ~~an OS Extension made up of the following features:~~
  - ~~a second (Process) stack, see *The SP registers* on page B1-8~~
  - ~~support of the `SVC` instruction and its associated SVCall exception, see *Overview of the exceptions supported* on page B1-13~~
  - ~~support of the PendSV interrupt described in *Overview of the exceptions supported* on page B1-13~~
  - support of the system timer - SysTick described in *System timer - SysTick* on page B3-13.
- a Debug Extension supporting a Debug state (halting debug) and associated control and configuration registers. See Part C on page C1-1 for more details.

ARMv6-M supports the privileged application/OS software model ~~when the OS Extension option is implemented. The architecture variant names for implementations with and without the OS Extension are as follows:~~

- ~~where the OS extension is not present, ARMv6-M~~
- ~~where the OS extension is present, ARMv6S-M.~~

Note

In deeply embedded systems, particularly at low cost/performance points, there is often no clear distinction between the operating system and the application(s). The software is designed as a single entity with no standardization of system interfaces or user versus privileged split. ARMv6-M correlates well with this programming model.

B1.2 ARMv6-M: a memory mapped architecture

ARMv6-M is a memory-mapped architecture, meaning physical addresses as well as processor registers are architecturally assigned to provide event entry points (vectors), system control and configuration. Exception handler entry points are maintained in a table of address pointers.

The address space `0xE0000000` to `0xFFFFFFFF` is reserved for system level use. The first 1MB of the system address space (`0xE0000000` to `0xE00FFFFF`) is reserved by ARM and known as the Private Peripheral Bus (PPB), with the rest of the address space (from `0xE0100000`) IMPLEMENTATION DEFINED with some memory attribute restrictions. See *The system address map* on page B3-2 for more details.

Within the PPB address space, a 4kB block in the range `0xE000E000` to `0xE000EFFF` is assigned for system control and known as the System Control Space (SCS). The SCS supports:

- CPU ID registers
- General control and configuration
- System handler support (for system interrupts and exceptions)
- A SysTick system timer ~~as part of the OS Extension option~~
- A Nested Vectored Interrupt Controller (NVIC), supporting up to 32 discrete external interrupts. All exceptions and interrupts share a common prioritization model
- Processor debug (optional in ARMv6-M)

See *System Control Space (SCS)* on page B3-5 for more details.

B1.3 System level operation and terminology overview

Several concepts are critical to the understanding of the system level architecture support.

B1.3.1 Modes, Privilege and Stacks

Mode, privilege and stack pointer are key concepts used in ARMv6-M.

Mode The microcontroller profile supports two modes (Thread and Handler modes). Handler mode is entered as a result of an exception. An exception return can only be issued in Handler mode.

Thread mode is entered on Reset, and can be entered as a result of an exception return.

Privilege ARMv6-M always executes in a privileged manner. Privileged execution has access to all resources.

Stack Pointer Two separate banked stack pointers exist ~~when the OS Extension is implemented. The Main Stack Pointer always exists, and the Process Stack pointer is supported by the option.~~ The Main Stack Pointer can be used in either Thread or Handler mode. The Process Stack Pointer can only be used in Thread mode. See *The SP registers* on page B1-8 for more details.

Table B1-1 shows the relationship between mode, privilege and stack pointer usage.

Table B1-1 Mode, privilege and stack relationship

Mode	Stack Pointer	Example (typical) usage model
Handler	Main	Exception handling
Handler	Process	Reserved combination (Handler always uses the Main stack)
Thread	Main	Execution of a privileged process/thread using a common stack in a system that only supports privileged access
Thread	Process	Execution of a privileged process/thread using a stack reserved for that process/thread in a system that only supports privileged access.

B1.3.2 Exceptions

An exception is a condition that changes the normal flow of control in a program. Exception behavior splits into two parts:

- Exception recognition when an exception event is generated and presented to the processor.
- Exception processing (activation) when the processor is executing an exception entry, exception return, or exception handler code sequence. Migration from exception recognition to processing can be instantaneous.

Exceptions can be split into four categories

Reset Reset is a special form of exception which terminates current execution in a potentially unrecoverable way when reset is asserted. When reset is de-asserted execution is restarted from a fixed point.

Supervisor call (SVCall)
An exception which is explicitly caused by the `SVC` instruction. A supervisor call is used by application code to make a system (service) call to an underlying operating system. The `SVC` instruction enables the application to issue a system call that requires privileged access to the system and will execute in program order with respect to the application. ARMv6-M also supports an interrupt driven service calling mechanism PendSV (see *Interrupts* in *Overview of the exceptions supported* on page B1-13 for more details).

Note

~~SVCall and PendSV are part of the OS Extension option.~~

Fault A fault is an exception which results from an error condition due to instruction execution or exception processing. Faults can be reported synchronously or asynchronously to the instruction which caused them. In general, faults are reported synchronously. The Imprecise BusFault is an asynchronous fault supported in the ARMv6-M profile.

A synchronous fault is always reported with the instruction which caused the fault. An asynchronous fault does not guarantee how it is reported with respect to the instruction which caused the fault. Faults are considered fatal in ARMv6-M (no fault status information is provided to assist recovery).

Interrupt An interrupt is an exception, other than a reset, fault or a supervisor call. All interrupts are asynchronous to the instruction stream. Typically interrupts are used by other elements within the system which wish to communicate with the processor, including software running on other processors.

Each exception has:

- a priority level
- an exception number
- a vector in memory which defines the entry-point (address) for execution on taking the exception. The associated code is described as the exception handler or the interrupt service routine (ISR).

Each synchronous exception, other than reset, is in one of three possible states:

- an Inactive exception is one which is not Pending or Active
- a Pending exception is one where the exception event has been generated, and which has not yet started being processed on the processor
- an Active exception is one whose handler has been started on a processor, but processing is not complete. An Active exception can be either running or pre-empted by a higher priority exception.

Asynchronous exceptions can be in one of the three possible states or both Pending and Active at the same time, where one instance of the exception is Active, and a second instance of the exception is Pending.

Priority Levels and Execution Pre-emption

All exceptions are assigned a priority level, the exception priority. Three exceptions have fixed values (Reset, NMI and HardFault), while the priority of SVCall, PendSV, SysTick and NVIC exceptions can be altered by software. In addition, the instruction stream executing on the processor has a priority level associated with it, the execution priority. A pending exception whose priority is higher than the execution priority will become active. In this case, the currently running instruction stream is pre-empted, and the exception that is taken is activated.

When an instruction stream is pre-empted by an exception other than reset, key context information is saved onto the stack automatically. Execution branches to the code pointed to by the exception vector that has been activated.

The execution priority can be boosted by software using the PRIMASK register provided for this purpose, otherwise it is the highest priority of all the exceptions that are active. See *Execution priority and priority boosting within the core* on page B1-16 for more details.

Exceptions can occur during the exception activation, for example as a result of a memory fault while pushing context information. The behavior of this case is described in *Exceptions on exception entry* on page B1-25.

Exception Return

When in handler mode, an exception handler can return. If the exception is both Active and Pending (a second instance of the exception has occurred while it is being serviced), it is re-entered or becomes Pending according to the prioritization rules. If the exception is Active only, it becomes Inactive. The key information[1] that was stacked is restored, and execution returns to the code pre-empted by the exception. The target of the exception return is determined by the Exception Return Link, a value stored in the link register on exception entry.

On an exception return, there can be a pending exception which is of sufficiently high priority that the pending exception will pre-empt the execution being returned to. This will result in an exception entry sequence immediately after an exception return sequence. This condition is referred to as chaining of the exceptions. Hardware can optimize chaining of exceptions to remove the need to restore and re-save the key context state, an optimization referred to as *tail-chaining*. See *Tail-chaining and exceptions on exception return* on page B1-26 for details.

Faults can occur during the exception return, for example as a result of a memory fault while popping previous state off the stack. The behavior in this and other cases is explained in *Derived exceptions* on page B1-26.

1. Where the exception is returning to Thread mode, the exception return can use different stack-based information, for example due to a context switch managed by an operating system.

B1.3.3 Execution State

ARMv6-M only executes Thumb instructions, both 16-bit and a few 32-bit instructions, and hence is always executing in Thumb state. Thumb state is indicated by an execution status bit (EPSR.T == 1) within the architecture, see *The special-purpose program status registers (xPSR)* on page B1-8. ARMv6-M is consistent with the software programming model and interworking support of additional execution states in other ARM architecture profiles. Setting EPSR.T to zero in ARMv6-M causes a fault when the next instruction executes, because all instructions in this state are UNDEFINED.

B1.3.4 Debug State

Debug state is entered when a core is configured to halt on a debug event, and a debug event occurs. See Chapter C1 *ARMv6-M Debug* for more details.

——— **Note** ———

Debug state is part of the Debug Extension for ARMv6-M.

————————————

B1.4 Registers

The ARMv6-M profile has the following registers closely coupled to the core:

- general purpose registers R0-R12
- 2 Stack Pointer registers, SP_main and SP_process (banked versions of R13)
- the Link Register, LR (R14)
- the Program Counter, PC
- status registers for flags, exception/interrupt level, and execution state bits
- a mask register (PRIMASK) associated with managing the prioritization scheme for exceptions and interrupts
- a control register (CONTROL) to identify the current stack.

All other registers described in this specification are memory mapped.

Note

Register access restrictions where stated apply to normal execution. Debug restrictions can differ, see *General rules applying to debug register access* on page C1-5, *Debug Core Register Selector Register (DCRSR)* on page C1-15 and *Debug Core Register Data Register (DCRDR)* on page C1-16.

B1.4.1 The SP registers

There are two stacks supported in ARMv6-M, each with its own (banked) stack pointer register.

- the Main stack – SP_main
- the Process stack – SP_process. ~~This stack is only available if the OS Extension is implemented, otherwise any access to SP_process is UNPREDICTABLE.~~

ARMv6-M treats bits [1:0] as RAZ/WI. Software should treat bits [1:0] as SBZP for maximum portability across other profiles.

The SP that is used by instructions which explicitly reference the SP is selected according to the function `LookUpSP()` described in *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-11.

The stack pointer that is used in exception entry and exit is described in the pseudocode sequences of the exception entry and exit, see *Exception entry behavior* on page B1-18 and *Exception return behavior* on page B1-21 for more details. SP_main is selected and initialized on reset, see *Reset behavior* on page B1-17.

B1.4.2 The special-purpose program status registers (xPSR)

Program status at the system level breaks down into three categories. They can be accessed as individual registers, a combination of any two from three, or a combination of all three using the `MRS` and `MSR` instructions.

- The Application Program Status Register APSR - User writeable flags. APSR handling of user writeable flags by the `MSR` and `MRS` instructions is consistent across ARMv6T2, ARMv6-M, and all ARMv7 profiles.

- The Interrupt Program Status Register IPSR – Exception Number (for current execution)
- The Execution Program Status Register EPSR - Execution state bits

The APSR, IPSR and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

Table B1-2 The xPSR register layout

	31	30	29	28		24		9		5	0
APSR	N	Z	C	V							
IPSR										0 or Exception Number	
EPSR						T		[a]			

a. While EPSR [9] is reserved, its associated bit location in memory for stacking xPSR context information is allocated to stack alignment support, see *Stack alignment on exception entry* on page B1-20.

The APSR is modified by flag setting instructions and used to evaluate conditional branch instructions. The flags (NZCV) are as described in *The Application Program Status Register (APSR)* on page A2-12. The flags are UNPREDICTABLE on reset.

The IPSR is written on exception entry and exit. It can be read using an `MRS` instruction. Writes to the IPSR by an `MSR` instruction are ignored. The IPSR Exception Number field is defined as follows:

- When in Thread mode, the value is 0.
- When in Handler mode, the value reflects the exception number as defined in *Exception number definition* on page B1-14.

The exception number is used to determine the currently executing exception and its entry vector (see *Exception number definition* on page B1-14 and *The vector table* on page B1-15).

On reset, the core is in Thread mode and the Exception Number field of the IPSR is cleared. As a result, the value 1 (the Reset Exception Number) is a transitory value, and not a valid IPSR Exception Number.

The EPSR contains the T-bit and cannot normally be read by software (for access in debug state see *Debug Core Register Data Register (DCRDR)* on page C1-16). All EPSR fields read as zero using an `MRS` instruction. `MSR` writes are ignored.

The EPSR T-bit supports the ARM architecture interworking model. ARMv6-M only supports execution of Thumb instructions, therefore it must always be maintained with the value T-bit == 1. Updates to the PC which comply with the Thumb instruction interworking rules must update the T-bit accordingly. The execution of an instruction with the EPSR T-bit clear will cause a HardFault.

The T-bit is initialized to 1 on reset (see *Reset behavior* on page B1-17 for details). All unused bits in the individual or combined registers are reserved.

B1.4.3 The special-purpose mask register

In ARMv6-M there is one special-purpose mask register, the exception mask register PRIMASK, that is used for priority boosting.

The format of the mask register is illustrated in Table B1-3.

Table B1-3 The special-purpose mask register

	31 8 7 1	0
PRIMASK	Reserved	PM

PRIMASK is cleared on reset and is accessible using the `MSR`/`MRS` instructions. Its function is explained in *Execution priority and priority boosting within the core* on page B1-16.

In addition:

- PRIMASK is set by the execution of the instruction: CPSID i
- PRIMASK is cleared by the execution of the instruction: CPSIE i.

B1.4.4 The special-purpose control register

The special-purpose CONTROL register is defined as follows:

- bit [0] reserved (UNK/SBZP)
- bit [1] defines the stack to be used
  - 0: SP_main is used as the current stack.
  - 1: For Thread mode, SP_process is used for the current stack. For Handler mode, this value is reserved.
  - Software can update bit [1] in Thread mode. Explicit writes from Handler mode are ignored.
  - The bit is updated on exception entry and exception return. See the pseudocode in *Exception entry behavior* on page B1-18 and *Exception return behavior* on page B1-21 for more details.
- bits [31:2] reserved.

CONTROL [1] is cleared on reset. The `MRS` instruction is used to read the register, and the `MSR` instruction is used to write the register.

An ISB barrier instruction is required to ensure a CONTROL register write access takes effect before the next instruction is executed.

B1.4.5 Reserved special-purpose register bits

All unused bits in special-purpose registers are reserved. `MRS` and `MSR` instructions that access reserved bits treat them as RAZ/WI. For future software compatibility, the bits are UNK/SBZP. Software should write them to zero when initializing the register for a new process, otherwise software should restore reserved bits when updating or restoring a special-purpose register.

B1.4.6 Special-purpose register updates and the memory order model

With the exception of writes to the CONTROL register, all changes to special-purpose registers from a `CPS` or `MSR` instruction are guaranteed:

- not to affect those instructions or preceding instructions in program order
- to be visible to all instructions that appear in program order after those changes.

B1.4.7 Register related definitions for pseudocode

Two register types are used in the system programmers' model pseudocode:

- 32- bit core registers
- 32-bit memory mapped registers.

See Appendix G *Register Index* for a list of ARMv6-M registers.

For bit fields associated with registers the convention adopted is to describe them as `<register_name>.<bitfield_name>` or by a specific bit reference. For example:

- AIRCR.SYSRESETREQ
- CONTROL [1]

Pseudocode details for ARM core register access in the Thumb instruction set

The following pseudocode supports access to the general-purpose registers for the Thumb instruction set operations defined in *Alphabetical list of ARMv6-M Thumb instructions* on page A6-14:

```
// The M-profile execution modes.

enumeration Mode {Mode_Thread, Mode_Handler};

// The names of the core registers. SP is a banked register.

enumeration RName {RName0, RName1, RName2, RName3, RName4, RName5, RName6,
                   RName7, RName8, RName9, RName10, RName11, RName12,
                   RNameSP_main, RNameSP_process, RName_LR, RName_PC};

// The physical array of core registers.
//
// _R[RName_PC] is defined to be the address of the current instruction.
// The offset of 4 bytes is applied to it by the register access functions.

array bits(32) _R[RName];
```

```
// LookUpSP()
// ==========

RName LookUpSP()
    RName SP;
    Mode CurrentMode;

    if CONTROL<1> == 1
        if CurrentMode==Mode_Thread then
            SP is RNameSP_process;
        else
            UNPREDICTABLE;
    else
        SP is RNameSP_main;
    return SP;


// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        result = _R[RName_PC] + 4;
    elsif n == 14 then
        result = _R[RName_LR]
    elsif n == 13 then
        LookUpSP();
        result = _R[SP];
    else
        result = _R[RName:n];
    return result;


// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    if n == 13 then
        LookUpSP();
        _R[SP] = value;
    else
        _R[RName:n] = value;
    return;


// BranchTo()
// ==========

BranchTo(bits(32) address)
    _R[RName_PC] = address;
    return;
```

B1.5 Exception model

The exception model is central to the architecture and system correctness in the ARMv6-M profile. The ARMv6-M profile is the same as ARMv7-M in using hardware saving and restoring of key context state on exception entry and exit, and using a table of vectors to determine the exception entry points. In addition, the exception categorization in the ARMv6-M profile is a subset of those provided in ARMv7-M.

B1.5.1 Overview of the exceptions supported

The following exceptions are supported by the ARMv6-M profile.

Reset Two levels of reset are supported by the ARMv7-M profile. The levels of reset control which register bit fields are forced to their reset values on the de-assertion of reset.

- Power-On Reset (POR) resets the core, System Control Space and debug logic.
- Local Reset resets the core and System Control Space except some fault and debug-related resources. For more details, see *Debug and reset* on page C1-9.

The Reset exception is permanently enabled, and has a fixed priority of -3.

NMI – Non Maskable Interrupt Non Maskable Interrupt is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2.

NMI can be set to the Pending state by software (see *Interrupt Control State Register (ICSR)* on page B3-8) or hardware.

HardFault HardFault is the generic fault that exists for all classes of fault that cannot be handled by any of the other exception mechanisms. HardFault is typically used for unrecoverable system failure situations, though this is not required, and some uses of HardFault might be recoverable. HardFault is permanently enabled and has a fixed priority of -1.

HardFault is used for all fault conditions on ARMv6-M.

SVCall This supervisor call handles the exception caused by the SVC instruction. ~~SVCall is part of the OS Extension option. Where implemented,~~ SVCall is permanently enabled and has a configurable priority, ~~otherwise the SVC instruction is UNDEFINED.~~

Interrupts The ARMv6-M profile supports two system level interrupts ~~as part of the OS Extension option~~ – PendSV for software generation of asynchronous system calls, and SysTick for a Timer integral to the ARMv6-M profile – along with up to 32 external interrupts. All interrupts have a configurable priority.

~~Where provided,~~ PendSV[1] is a permanently enabled interrupt, controlled using ICSR.PENDSVSET and ICSR.PENDSVCLR (see *Interrupt Control State Register (ICSR)* on page B3-8). SysTick can not be disabled.

1. A service (system) call is used by an application which requires a service from an underlying operating system. The service call associated with PendSV executes when the interrupt is taken. For a service call which executes synchronously with respect to program execution use the SVC instruction (the SVCall exception).

Note

While hardware generation of a SysTick event can be suppressed, ICSR.PENDSTSET and ICSR.PENDSTCLR (see *Interrupt Control State Register (ICSR)* on page B3-8) are always available to software.

All external interrupts can be disabled. Interrupts can be set to or cleared from the Pending state by software, and interrupts other than PendSV can be set to the Pending state by hardware.

See *Fault behavior* on page B1-29 for a definitive list of all the possible causes of faults. ARMv6-M does not support run-time fault status register bits to identify the faults (only a Debug Fault Status Register in the Debug Extension).

B1.5.2 Exception number definition

All exceptions have an associated exception number as defined in Table B1-4.

Table B1-4 Exception numbers

Exception number	Exception
1	Reset
2	NMI
3	HardFault
4-10	Reserved
11	SVCall ~~(OS Extension only)~~
12-13	Reserved
14	PendSV ~~(OS Extension only)~~
15	SysTick ~~(OS Extension only)~~
16	External Interrupt(0)
…	…
16 + N	External Interrupt(N)

B1.5.3 The vector table

The vector table contains the initialization value for the stack pointer on reset, and the entry point addresses for all exception handlers. The exception number ~~(on page B1-14~~*Exception number definition* on page B1-14) defines the order of entries in the vector table associated with exception handler entry as illustrated in Table B1-5.

Table B1-5 Vector table format

word offset	Description – all pointer address values
0	SP_main (reset value of the Main stack pointer)
Exception Number	Exception using that Exception Number

For ARMv6-M, the vector table base address is fixed at 0x00000000[1]. The entry at offset 0 is used to initialize the value for SP_main, see *The SP registers* on page B1-8. All other entries must have bit [0] set, as the bit is used to define the EPSR T-bit on exception entry (see *Reset behavior* on page B1-17 and *Exception entry behavior* on page B1-18 for details). On exception entry, if bit [0] of the associated vector table entry is clear, execution of the first instruction will cause a HardFault.

B1.5.4 Exception priorities and pre-emption

The priority algorithm treats lower numbers as taking higher precedence, that is, the lower the assigned value the higher the priority level. Exceptions assigned the same priority level adopt a fixed priority order for selection within the architecture according to their exception number.

Reset, non-maskable interrupts (NMI) and HardFault execute at fixed priorities of -3, -2, and -1 respectively. All other exception priorities can be set under software control and are cleared on reset.

The priorities of all exceptions are set in registers within the System Control Space (specifically, registers within the system control block and NVIC).

When multiple exceptions have the same priority number, the pending exception with the lowest exception number takes precedence. Once an exception is active, only exceptions with a higher priority (lower priority number) can pre-empt it.

The priority field is an 8-bit field. ARMv6-M supports four priority levels, using the two most significant bits. The other six bits in the priority field are reserved.

If the priority of an exception is changed when it is Active or enabled[2], the effect is UNPREDICTABLE in ARMv6-M.

1. For ARMv6-M, the Vector Table Offset Register defined in ARMv7-M is reserved with respect to software, and defined as RAZ/WI with respect to hardware.
2. The *not enabled* condition does not apply to the permanently enabled SVCall or PendSV exceptions. It applies to all cases where the configurable priority exception can be disabled in software.

Priority grouping

Priority grouping is a method of assigning a configurable number of the least significant priority bits such that they are ignored when determining the highest priority pending exception. The effect is to group the affected exceptions under a common priority defined by the remaining (most significant) priority bits. Configurable priority grouping is not supported in ARMv6-M meaning all supported priority bits are always used for the (group) priority when detecting the highest pending exception. The PRIGROUP field of the Application Interrupt and Reset Control Register (AIRCR) is a RAZ bitfield, meaning all (2) implemented bits of the priority are always used for the group priority.

When two pending exceptions have the same group priority, the lower pending exception number has priority over the higher pending number as part of the priority precedence rule.

The group priorities of Reset, NMI and HardFault are -3, -2, and -1 respectively.

Execution priority and priority boosting within the core

The execution priority is defined to be the highest priority formulated from a set of values:

- the priorities of all Active exceptions (lowest priority number)
- the impact of PRIMASK and priority boosting.

Note

Note: If the priority of an active exception is changed it can affect the execution priority. Therefore, the execution priority can be different from the priority of the running exception. This ensures that dynamic priority management avoids priority inversion from a new exception with respect to the active exception stack.

Setting the mask bit in the PRIMASK register raises the execution priority to 0. This prevents all exceptions with configurable priority from activating, other than through the fault escalation mechanism (see *Priority escalation*). This also has a special impact on `WFI` (see *WFI* on page A6-110).

Priority boosting only affects the group priority. It has no effect on the sub-priority. The sub-priority is only used to sort pending exception priorities, and does not affect active exceptions.

Note

In ARMv6-M, Thread mode has an implicit priority of the lowest priority in the system. Thread mode executes when there are no pending or active exceptions.

Priority escalation

When the group priority of a supervisor call (SVCall) is lower than or equal to the currently executing group priority[1], inhibiting normal pre-emption, a HardFault exception is taken. This is known as *priority escalation*.

A fault which is escalated to a HardFault retains the `ReturnAddress()` behavior of the original fault. See the pseudocode definition of `ReturnAddress()` in *Exception entry behavior* on page B1-18 for more details. For the behavior of the affected exceptions occurring when the currently executing group priority is that of a HardFault or higher, see *Unrecoverable exception cases* on page B1-30.

B1.5.5 Reset behavior

The assertion of reset causes the current execution state to be abandoned without being saved. On the de-assertion of reset, all registers controlled by the reset assertion contain their reset values, and the following actions are performed.

For global declarations see *Register related definitions for pseudocode* on page B1-11.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxE-22.

```
// TakeReset()
// ============

~~integer NestedActivation;~~           ~~/* used for Handler => Thread check when value == 1 */~~
bit ExceptionActive[*];              /* conceptual array of 1-bit values for all exceptions */
bits(32) vectortable = '00':VTOR<29:7>:'0000000';
Mode CurrentMode;

TakeReset()
    R[0..12] = bits(32) UNKNOWN;
    SP_main = MemA[vectortable,4] & 0xFFFFFFFC;
    SP_process = ((bits(30) UNKNOWN):'00');
    LR = bits(32) UNKNOWN;              /* value must be initialised by software */
    tmp = MemA[vectortable+4,4]
    PC = tmp AND 0xFFFFFFFE;            /* address of reset service routine */
    tbit = tmp<0>;
    CurrentMode = Mode_Thread;
    APSR = bits(32) UNKNOWN;            /* flags UNPREDICTABLE from reset */
    IPSR<5:0> = 0x0;                    /* Exception Number cleared */
    EPSR.T = tbit;                      /* T bit set from vector */
    EPSR.ICI<7:0> = 0x0;                /* ICI bits cleared */
    PRIMASK<0> = '0';                   /* priority mask cleared at reset */
    BASEPRI<7:0> = 0x0;                 /* base priority disabled at reset */
    CONTROL<1> = '0';                   /* current stack is Main */
    CONTROL<0> = '0';                   /* Thread is privileged */
    ResetSCSRegs();                     /* catch-all function for System Control Space reset */
    ~~NestedActivation = 0x0;~~             ~~/* initialised value for base Thread */~~
    ExceptionActive[*] = '0';           /* all exceptions Inactive */
    ClearEventRegister()                  /* see WFE instruction for more details */
```

ExceptionActive[*] is a conceptual array of active flag bits for all exceptions (fixed priority system exceptions, configurable priority system exceptions, and external interrupts). For ARMv6-M, NestedActivation is conceptual too.

1. Applies when the currently executing group priority is less than HardFault. If an exception occurs at a currently executing group priority of HardFault or higher, the behavior is defined in *Unrecoverable exception cases* on page B1-30.

B1.5.6 Exception entry behavior

On pre-emption of an instruction stream, context state is saved by the hardware onto a stack pointed to by one of the SP registers (see *The SP registers* on page B1-8). The stack that is used depends on the mode of the processor at the time of the exception.

The stacked context supports the ARM Architecture Procedure Calling Standard (AAPCS). The support allows the exception handler to be an AAPCS-compliant procedure.

A full-descending stack format is used, where the stack pointer is decremented ~~immediately~~ before storing a 32-bit word (when pushing context) onto the stack, and incremented after reading a 32-bit word (popping context) from the stack. Eight 32-bit words are saved in descending order, with respect to their address in memory, as listed:

xPSR, `ReturnAddress()`, LR (R14), R12, R3, R2, R1, and R0

The exception entry pseudocode is:

```
// ExceptionEntry()
// ================

// NOTE: PushStack() can abandon memory accesses if a fault occurs during the stacking
//       sequence.
//       Exception entry is modified according to the behavior of a derived exception.

PushStack();
ExceptionTaken(ExceptionType);    // ExceptionType is encoded as its exception number
```

For global declarations see *Register related definitions for pseudocode* on page B1-11.
For a definition of `ExceptionActive[*]` and `NestedActivation` see *Reset behavior* on page B1-17.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxE-22.

The `PushStack()` and `ExceptionTaken()` pseudo-functions are defined as follows:

```
// PushStack()
// ===========

PushStack()
    if CONTROL<1> == '1' AND CurrentMode == Mode_Thread then
        frameptralign = SP_process<2>;
        SP_process = (SP_process - 0x20) AND NOT(ZeroExtend('100',32));
        frameptr = SP_process;
    else
        frameptralign = SP_main<2>;
        SP_main = (SP_main - 0x20) AND NOT(ZeroExtend('100',32));
        frameptr = SP_main;
       /* only the stack locations, not the store order, are architected */
    MemA[frameptr,4]      = R[0];
    MemA[frameptr+0x4,4]  = R[1];
    MemA[frameptr+0x8,4]  = R[2];
    MemA[frameptr+0xC,4]  = R[3];
    MemA[frameptr+0x10,4] = R[12];
    MemA[frameptr+0x14,4] = LR;
    MemA[frameptr+0x18,4] = ReturnAddress();
    MemA[frameptr+0x1C,4] = (xPSR<31:10>:frameptralign:xPSR<8:0>);
    if CurrentMode==Mode_Handler then
        LR = 0xFFFFFFF1;
    else
        if CONTROL<1> == '0' then
            LR = 0xFFFFFFF9;
        else
            LR = 0xFFFFFFFD;
    return;


// ExceptionTaken()
// ================

ExceptionTaken(ExceptionNumber)

    bit tbit;
    bits(32) tmp;

    R[0..3] = bits(32) UNKNOWN;
    R[12] = bits(32) UNKNOWN;
    tmp = MemA[VectorTable+4*ExceptionNumber,4];
    PC = tmp AND 0xFFFFFFFE;
    tbit = tmp<0>;
    CurrentMode = Mode_Handler;
    APSR = bits(32) UNKNOWN;                // Flags UNPREDICTABLE due to other activations
    IPSR<5:0> = ExceptionNumber                 // ExceptionNumber set in IPSR
    EPSR.T = tbit;                              // T-bit set from vector
    /* PRIMASK and BASEPRI unchanged on exception entry */
    CONTROL<1> = '0';                           // current Stack is Main
    /* CONTROL<0> unchanged */
    ~~NestedActivation = NestedActivation + 1;~~
    ExceptionActive[ExceptionNumber]= '1';
    SCS_UpdateStatusRegs();                     // update SCS registers as appropriate
    SetEventRegister()                          // see WFE instruction for more details
    InstructionSynchronizationBarrier();
```

For more details on the registers with UNKNOWN values, see *Exceptions on exception entry* on page B1-25

For updates to system status registers, see section *System Control Space (SCS)* on page B3-5.
`ReturnAddress()` is the address to which execution will return after handling of the exception:

```
// ReturnAddress()
// ===============

ReturnAddress() returns the following values based on the exception cause
   // NOTE: ReturnAddress() is always halfword aligned - bit<0> is always zero

// Exception Type          Address returned
// ==============          ================

// NMI:                    Address of Next Instruction to be executed
// HardFault (precise):    Address of the Instruction causing fault
// HardFault (imprecise):  Address of Next Instruction to be executed
// SVC:                    Address of the Next Instruction after the SVC
// IRQ:                    Address of Next Instruction to be executed after an interrupt
```

Note

IRQ includes SysTick and PendSV~~ where implemented~~.

~~Where the OS Extension is not supported, the SVC instruction is UNDEFINED, it causes a precise HardFault and ReturnAddress() is the address of the SVC instruction.~~ Where the priority of an SVCall is escalated, this causes a HardFault with the ReturnAddress() as stipulated for the SVC related exception.

B1.5.7 Stack alignment on exception entry

ARMv6-M mandates that all exceptions are entered with 8-byte stack alignment. As exceptions can occur on any instruction boundary, it is possible that the current stack pointer is not 8-byte aligned when an exception activates.

The AAPCS requires that the stack-pointer be 8-byte aligned on entry to a conforming function[1]. Since it is anticipated that exception handlers will be written as AAPCS conforming functions, the system must ensure natural alignment of the stack for all arguments passed. The 8-byte alignment requirement is guaranteed in hardware in ARMv6-M.

Theory of operation

On an exception entry, the stack pointer (SP_main or SP_process) in use before the exception entry is forced to have 8-byte alignment by adjusting its alignment as part of the exception entry sequence. The xPSR that is saved as part of the exception entry sequence records the alignment of this stack pointer prior to the exception entry sequence. The alignment status is merged and stored to memory as bit [9] of the xPSR (a reserved bit within the xPSR) in the saved context information.

On an exception exit, the stack pointer returned to takes its alignment from the value recovered from bit [9] of the xPSR in the restored context from the exception exit sequence. This reverses the forced stack alignment performed on the exception entry.

See *Exception entry behavior* on page B1-18 and *Exception return operation* on page B1-23 for pseudocode details of the effect of the stack alignment feature on exception entry and exception return.

Note

Stack pointer alignment on exception exit is architecturally defined as an OR function. If the exception exit sequence is started with a stack pointer which is only 4 byte aligned, then this change has no effect.

1. The AAPCS requires conforming functions to preserve the natural alignment of primitive data of size 1, 2, 4, and 8 bytes. In return, conforming code is permitted to rely on that alignment. To support unqualified reliance the stack-pointer must in general be 8-byte aligned on entry to a conforming function. If a function is entered directly from an underlying execution environment, that environment must accept the stack alignment obligation in order to give an unqualified guarantee that conforming code can execute correctly in all circumstances.

In the event that the exception exit causes a derived exception, it is IMPLEMENTATION DEFINED whether the derived exception is entered with the same stack alignment as was in use before the exception exit sequence started.

A side-effect of this feature is that the amount of stack used on exception entry becomes a function of the alignment of the stack at the time that the exception is entered. As a result, the average and worst case stack usage will increase. The worst case increase is 4 bytes per exception entry.

Maintaining the stack alignment information in an unused bit within the saved xPSR makes the feature transparent to context switch code within operating systems, provided that the reserved status of unused bits within the xPSR has been respected.

Compatibility

Some operating systems can avoid saving and restoring R14 when switching between different processes in Thread mode if it is known that the values held in an EXC_RETURN value are invariant between the different processes. This provides a small improvement in context switch time, but at the cost of future compatibility. The stack alignment feature does not affect the EXC_RETURN value. ARM does not guarantee that this software optimization will be possible in future revisions of the ARMv6-M architecture, and recommends for future compatibility that the R14 value is always saved and restored on a context switch as part of the thread's context.

B1.5.8 Exception return behavior

Exception returns occur when one of the following instructions loads a value of `0xFXXXXXXX` into the PC while in Handler mode:

- `POP` which includes loading the PC.
- `BX` with any register.

When used in this way, the value written to the PC is intercepted and is referred to as the EXC_RETURN value.

EXC_RETURN<~~28~~:4> are reserved with the special condition that all bits should be written as one or preserved from exception entry. Values other than all 1s are UNPREDICTABLE. EXC_RETURN<3:0> provide return information as defined in Table B1-6.

Table B1-6 Exception return behavior

EXC_RETURN<3:0>	
0b0001	Return to Handler Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b1001	Return to Thread Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b1101~~[a]~~	Return to Thread Mode; Exception return gets state from the Process stack; On return execution uses the Process Stack.
Unused	Reserved

~~a. OS Extension only, otherwise this value is reserved.~~

Reserved entries in this table are UNPREDICTABLE.

If an EXC_RETURN value is loaded into the PC when in Thread mode, or from the vector table, or by any other instruction, the value is treated as an address, not as a special value. This address range is defined to have eXecute Never (XN) permissions, and will result in a HardFault exception.

Integrity checks on exception returns

Incorrect exception return information could be inconsistent with the state of execution which must be held in processor hardware or other state stored by the exception mechanisms.

ARMv6-M does not support integrity checking on exception return. An irregular EXC_RETURN or inconsistent xPSR value read from the stack is considered a fatal, unrecoverable error. The effect of an irregular EXC_RETURN is UNPREDICTABLE, however, it is recommended that cases which can be guaranteed to cause a HardFault or a lockup situation are documented as if they are IMPLEMENTATION DEFINED. In the case of a memory fault while recovering the xPSR:

- if the Main stack (SP_main) is used for the exception return, the processor will lockup with a HardFault
- if the Process stack (SP_process) is used for the exception return, a Pending HardFault exception is generated.

Exception return operation

For global declarations see *Register related definitions for pseudocode* on page B1-11.
For ExceptionTaken() see *Exception entry behavior* on page B1-18.
For a definition of ExceptionActive[*] ~~and NestedActivation~~ see *Reset behavior* on page B1-17.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxE-22.

```
// ExceptionReturn()
// =================

ExceptionReturn(bits(28) EXC_RETURN)
    assert CurrentMode == Mode_Handler;
    if !IsOnes(EXC_RETURN<27:4>) then UNPREDICTABLE;

    integer ReturningExceptionNumber = UInt(IPSR<5:0>);

    if ExceptionActive[ReturningExceptionNumber] == '0' then
        UNPREDICTABLE;                // returning from an inactive handler
    else
        case  EXC_RETURN<3:0> of
            when '0001'                               // return to Handler
                if NestedActivation == 1 then
                    UNPREDICTABLE;        // return to Handler exception mismatch
                else
                    frameptr = SP_main;
                    CurrentMode = Mode_Handler;
                    CONTROL<1> = '0';
            when '1001'                               // returning to Thread using Main stack
                if NestedActivation != 1 then
                    UNPREDICTABLE;        // return to Thread exception mismatch
                else
                    frameptr = SP_main;
                    CurrentMode = Mode_Thread;
                    CONTROL<1> = '0';
            when '1101'                               // returning to Thread using Process stack
                if NestedActivation != 1 then
                    UNPREDICTABLE;        // return to Thread exception mismatch
                else
                    frameptr = SP_process;
                    CurrentMode = Mode_Thread;
                    CONTROL<1> = '1';
            otherwise
                UNPREDICTABLE;            // illegal EXC_RETURN

        DeActivate(ReturningExceptionNumber);
        PopStack(frameptr);
        if CurrentMode==Mode_Handler AND IPSR<5:0> == '000000000' then
            UNPREDICTABLE;                // return IPSR is inconsistent
        if CurrentMode==Mode_Thread AND IPSR<5:0> != '000000000' then
            UNPREDICTABLE;                // return IPSR is inconsistent
        SetEventRegister()                // see WFE instruction for more details
        InstructionSynchronizationBarrier();
```

The DeActivate() and PopStack() pseudo-functions are defined as follows:

```
// DeActivate()
// ============

DeActivate(integer ReturningExceptionNumber)
    ExceptionActive[ReturningExceptionNumber] = '0';
    /* PRIMASK and BASEPRI unchanged on exception exit */
    ~~NestedActivation = NestedActivation - 1;~~
    return;

// PopStack()
// ==========

PopStack(bits(32) frameptr)   /* only stack locations, not the load order, are architected */
    R[0]  = MemA[frameptr,4];
    R[1]  = MemA[frameptr+0x4,4];
    R[2]  = MemA[frameptr+0x8,4];
    R[3]  = MemA[frameptr+0xC,4];
    R[12] = MemA[frameptr+0x10,4];
    LR    = MemA[frameptr+0x14,4];
    PC    = MemA[frameptr+0x18,4];         // UNPREDICTABLE if the new PC not halfword aligned
    psr   = MemA[frameptr+0x1C,4];
    case  EXC_RETURN<3:0> of
        when '0001'                         // returning to Handler
            SP_main = (SP_main + 0x20) OR (psr<9> <<2);
        when '1001'                         // returning to Thread using Main stack
            SP_main = (SP_main + 0x20) OR (psr<9> <<2);
        when '1101'                         // returning to Thread using Process stack
            SP_process = (SP_process + 0x20) OR (psr<9> <<2);
    APSR<31:28> = psr<31:28>;               // valid APSR bits loaded from memory
    IPSR<5:0> = psr<5:0>;                   // valid IPSR bits loaded from memory
    EPSR<24> = psr<24>;                     // valid EPSR bits loaded from memory
    return;
```

B1.5.9 Exceptions in single-word load operations

In order to comply with the *base-restore exception model*, single-word load instructions must not update the destination register when a fault occurs during execution. By example, this allows replay of the following instruction:

```
LDR R0, [R2, R0];
```

B1.5.10 Exceptions in LDM and STM operations

In ARMv6-M, it is IMPLEMENTATION DEFINED if interrupts are taken during the execution of a multi-word instruction (LDM, STM, PUSH or POP). If an interrupt is taken during a multi-word instruction, the instruction is abandoned, and if the return from the interrupt re-executes the instruction, the multi-word instruction is restarted. In the same way, if a precise HardFault occurs during the execution of a multi-word instruction, the instruction is abandoned, and if the return from the exception re-executes the instruction, the multi-word instruction is restarted. In all cases, instructions that are restarted perform all the memory accesses specified by the instruction, repeating memory accesses performed by the original execution of the instruction before it was abandoned.

To support instruction replay, the LDM, STM, PUSH and POP instructions are required to restore/maintain the base register when an exception occurs (no base register writeback update) during execution.

Note

Because interrupts might be taken during the execution of multi-word instructions, these instructions must not be used to access regions of memory if it is unacceptable to repeat a memory access, see *Device memory attribute* on page A3-12 and *Strongly Ordered memory attribute* on page A3-13.

Load multiple and PC in load list

For the ARM architecture in general, the case of LDM with PC in the register list is defined to be unordered, allowing the registers to be loaded in a different order than the register mask implies. The usual use is to allow the PC to be loaded first.

For ARMv6-M, if the PC was loaded early and the LDM instruction is restartable, the PC presented to the exception entry sequence must be restored such that the return address from the exception taken is to the LDM instruction address; it is then loaded again when the LDM restarts.

B1.5.11 Exceptions on exception entry

During exception entry other exceptions can occur, either because of a fault on the operations involved in exception entry, or because of the arrival of an asynchronous exception, an interrupt, which is of higher priority than the exception that caused the entry sequence in progress.

Late arriving exceptions

The ARMv6-M profile does not specify the point at which the arrival of an asynchronous exception is recognized during an exception entry. However, in order to support implementations with very low interrupt latencies, the ARMv6-M profile provides some facilities to permit high priority interrupts arriving during an exception entry to activate during the exception entry, and for the entry sequence not to be repeated.

When an asynchronous interrupt activates during the exception entry sequence, the exception that caused the exception entry sequence is known as the original exception. The exception caused by the interrupt is known as the secondary exception.

It is permissible in this case for the exception entry sequence that was started by the original exception to be used by the secondary exception. The original exception is taken after the secondary exception has returned. This is referred to as late-arrival pre-emption.

For a late arrival pre-emption, the secondary exception (interrupt, fault or supervisor call) is entered and the original exception is left in the Pending state.

It is IMPLEMENTATION DEFINED what conditions, if any, lead to late arrival pre-emption. Late arrival pre-emption can only occur when the secondary exception is of higher priority than the original exception. Where late arrival exceptions are supported:

```
// LateArrival()
// =============
LateArrival()

    // xEpriority: the lower the value, the higher the priority

    integer OEpriority;  // original exception group priority
    integer SEpriority;  // secondary exception group priority
    integer OEnumber;    // ExceptionNumber for OE
    integer SEnumber;    // ExceptionNumber for SE


    if (SEpriority < OEpriority) then
        ExceptionTaken(SEnumber);        // secondary exception taken
    else
        ExceptionTaken(OEnumber);        // original exception taken
```

~~For `ExceptionTaken()` and `PushStack()` see *Exception entry behavior* on page B1-18.~~

Derived exceptions

Where an exception entry sequence itself causes a fault, the exception that caused the exception entry sequence is known as the original exception. The fault that is caused by the exception entry sequence is known as the derived exception. The code stream that was running at the time of the original exception is known as the pre-empted code whose execution priority is the pre-empted priority.

The following HardFault faults can occur as derived exceptions during exception entry:

- a memory fault on the writes to the stack memory as part of the exception entry
- a memory fault on reading the vector for exception entry.

If the pre-empted group priority is lower than HardFault, the HardFault exception is set to Pending and the exception taken in accordance with the prioritization rules for pending exceptions. In this case, it is permissible for the HardFault exception to be treated as a late-arrival exception. If the pre-empted group priority is a HardFault (a priority value of -1), a lockup occurs (see *Unrecoverable exception cases* on page B1-30).

Derived exceptions that occur during an NMI entry sequence (group priority value of -2) cannot cause pre-emption.

B1.5.12 Tail-chaining and exceptions on exception return

During exception return, other exceptions can affect behavior, either because of a fault on the operations involved in exception return, or because of an asynchronous exception that is of higher priority than the priority level being returned to during the exception return. The asynchronous exception can be already Pending or arrive during the exception return.

The target of the exception return is described by the Exception Return Link. The target priority is the highest priority active exception, excluding the exception being returned from, or the boosted priority set by the PRIMASK register, whichever is higher.

Where the recognition of interrupts during an exception return is not supported, a HardFault caused by the exception return is the only exception that can occur at this time.

Derived exceptions

Where an exception return sequence causes a fault exception, the exception caused by the exception return sequence is known as the derived exception.

During an exception return, a memory fault on the reads from the stack memory will cause a HardFault derived exception. The restored state including the target priority can not be guaranteed, meaning that either a HardFault exception is taken or lock-up occurs, see *Unrecoverable exception cases* on page B1-30.

Tail-chaining

Tail-chaining is the optimization of an exception return and an exception entry so that the loads and stores of the key context state can be eliminated.

Tail-chaining is used for two reasons:

- For handling derived exceptions
- As an optimization that implementations are permitted to use to improve interrupt response when there is a Pending exception which has a higher group priority than the target group priority. In this case, the architected behavior is that the Pending exception will be taken immediately on exception return, and tail-chaining permits the optimization of the return and entry sequences.

In the tail-chaining optimization, the exception return and exception entry sequences are combined to form the following sequence, where the ReturningExceptionNumber is the number of the exception being returned from, and the ExceptionNumber is the number of the exception being tail-chained to. EXC_RETURN is the EXC_RETURN value that caused the original exception return to start.

For `ExceptionTaken()` see *Exception entry behavior* on page B1-18.
For `DeActivate()` see *Exception return behavior* on page B1-21.

```
// TailChain()
// ===========

TailChain(bits(28) EXC_RETURN)
    assert CurrentMode == Mode_Handler;
    if !IsOnes(EXC_RETURN<27:4>) then UNPREDICTABLE;

    integer ReturningExceptionNumber = UInt(IPSR<8:0>);
    LR = 0xF0000000 + EXC_RETURN;
    DeActivate(ReturningExceptionNumber);
    ExceptionTaken(ExceptionNumber);

    /* NestedActivation is effectively unchanged by a tail-chain */
```

Use of tail-chaining as an optimization for pending exceptions

The use of tail-chaining as an optimization for performing exception returns when there are Pending exceptions has a behavior which is different from simply performing the exception return, followed by the Pending exception entry. The difference in behavior is that many of the derived exceptions that could occur as a result of the exception return and the exception entry might not occur. Instead, these derived exceptions will occur when the pending exception is returned from.

Late arrival pre-emption and tail-chaining during exception returns

ARMv6-M does not specify the point at which arrival of asynchronous exceptions are recognized during an exception. ARMv6-M permits exceptions of a higher priority than the priority of the exception to be tail-chained to, to be entered in place of that exception being tail-chained to, using late-arrival pre-emption. It is IMPLEMENTATION DEFINED what conditions, if any, lead to late arrival pre-emption.

Late-arrival pre-emption can occur during a tail-chaining execution sequence due to a derived exception on an exception return. The derived exception is marked as Pending when a late-arrival pre-emption of the derived exception occurs.

B1.5.13 Exception status and control

The System Control Block within the System Control Space (*The System Control Block (SCB)* on page B3-7) provides the register support required to manage the exception model. The registers break down into the following categories:

- General system configuration, status and control
  - Interrupt Control State Register – see *Interrupt Control State Register (ICSR)* on page B3-8
  - Application Interrupt and Reset Control Register – see *Application Interrupt and Reset Control Register (AIRCR)* on page B3-9
  - System Handler Priority Registers – see *System Handler Priority Register 2 (SHPR2)* on page B3-11, and *System Handler Priority Register 3 (SHPR3)* on page B3-11
- SysTick support (~~OS Extension only)~~ – see *System timer - SysTick* on page B3-13
- NVIC support – see *Nested Vectored Interrupt Controller (NVIC)* on page B3-17

For ARMv6-M, support of stack alignment (see *Stack alignment on exception entry* on page B1-20), unaligned access trapping (see *Alignment support* on page A3-3) and priority grouping (see *Priority grouping* on page B1-16) are fixed in hardware with no programming interface.

The Interrupt Control State Register (*Interrupt Control State Register (ICSR)* on page B3-8) provides the ability to set the Pending state in software for NMI, SysTick and PendSV. It also provides the ability to clear the Pending state in software for SysTick and PendSV, and provides status information of pending and active exceptions.

The Application Interrupt and Reset Control Register (*Application Interrupt and Reset Control Register (AIRCR)* on page B3-9) provides endian status for data accesses, and reset control, see *Control of the Endian Mapping in ARMv6-M* on page A3-5 and *Reset management* on page B1-32 respectively for more details.

The System Handler Priority Registers (*System Handler Priority Register 2 (SHPR2)* on page B3-11 and *System Handler Priority Register 3 (SHPR3)* on page B3-11) provide the ability to program the priority of SVCall, SysTick and PendSV.

The NVIC registers (*NVIC register support in the SCS* on page B3-19) provide the following functions for external interrupts:

- enabling and disabling
- setting and clearing the Pending state
- programming the priority.

Note

Interrupts can become Pending when the associated interrupt is disabled. Enabling an interrupt allows a Pending interrupt to activate.

B1.5.14 Fault behavior

In accordance with the ARMv6-M exception priority scheme, a fault causes one of the following:

- a HardFault exception
- lock-up in the case of a fault arising while executing a HardFault or NMI, as described in *Unrecoverable exception cases* on page B1-30.

Faults are considered fatal in ARMv6-M. with no fault status information saved. The fault handler will return according to the rules defined in `ReturnAddress()`, see *Exception entry behavior* on page B1-18 for details.

Table B1-7 lists all faults. The information provided includes the cause and exception taken.

Table B1-7 List of supported faults

Fault Cause	Fault exception	Notes
Vector Read error	HardFault	Bus error returned when reading the vector table entry.
Fault escalation See *Priority escalation* on page B1-16 for more information.	HardFault	An SVCall occurred, and the handler group priority is lower or equal to the execution group priority. ~~(OS Extension only)~~
Memory fault on exception entry stack memory operations	HardFault	Failure when saving context via hardware – bus error returned.
Memory fault on exception return stack memory operations	HardFault	Failure when restoring context via hardware – bus error returned.

Table B1-7 List of supported faults (continued)

Fault Cause	Fault exception	Notes
Memory fault on instruction access (precise)	HardFault	Bus error on an instruction fetch or attempt to execute from memory marked as XN.
Precise error on data access	HardFault	Precise bus error due to an explicit memory access.
Imprecise error on data access	HardFault	Imprecise bus error due to an explicit memory access.
No Coprocessor	HardFault	No coprocessor support in ARMv6-M
Undefined Instruction	HardFault	Unknown instruction
Attempt to execute an instruction when EPSR.T==0	HardFault	Attempt to execute in an invalid EPSR state (e.g. after a BX type instruction has changed state). This includes state change after entry to or return from exception, as well as from inter-working instructions.
unaligned load or store	HardFault	This will occur when any load-store instruction attempts to access a non-aligned location.
For debug related faults – see Chapter C1 *ARMv6-M Debug*		

Fault status and address information

The ARMv6-M System Control Space includes fault status associated with debug only.

- A Debug Fault Status register – see *Debug Fault Status Register (DFSR)* on page B3-12 and Chapter Chapter C1 *ARMv6-M Debug* for more details

B1.5.15 Unrecoverable exception cases

The ARMv6-M and ARMv7-M profiles generally assume that when the processor is running at priority -1 or above, any faults or supervisor calls that occur are fatal and are entirely unexpected. While ARMv6-M does not provide fault status information to the HardFault handler, it does allow the handler to perform an exception return and continue execution where software is able to recover from the fault situation.

The standard exception entry mechanism does not apply where a fault or supervisor call occurs at a priority of -1 or above. ARMv7-M handles most of these cases using a mechanism called *lock-up*, otherwise the condition becomes Pending or is ignored. ARMv6-M uses lock-up in all its supported cases. Lock-up suspends normal instruction execution and enters lock-up state. When in lock-up state, the behavior is:

- the processor repeatedly fetches the same instruction from the fixed address 0xFFFFFFFF
- the instruction fetched is not a valid instruction (the memory is marked XN), and is not executed.

It is strongly recommended that implementations provide an external signal which indicates that the lock-up state has been entered to allow external mechanisms to react.

The lock-up state can be exited in one of 3 ways:

- If locked up at priority -1 and an NMI exception occurs, the NMI will be activated as normal. The NMI return link will be the address used for the lock-up state.
- A System reset occurs. This will exit lock-up state and reset the system as normal.
- A halt command from a halt mode debug agent is issued. The core will enter Debug state with the PC set to the same value as is used for return context. See Table B1-8 for details.

Table B1-8 outlines the behavior of all lock-up conditions.

Table B1-8 Lock-up conditions

Fault cause	Occurrence	Behavior	Lock-up Address
VECTABLE read error at reset	Cannot read vector table for SP or PC at reset	Lock-up at priority -1	`0xFFFFFFFF`
VECTABLE read error on NMI entry	Cannot read NMI vector	Lock-up at priority -2	`0xFFFFFFFF`
VECTABLE read error on HardFault entry	Cannot read HardFault vector	Lock-up at priority -1	`0xFFFFFFFF`
Memory Fault – Instruction	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFF`
Memory Fault – Imprecise Data	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFF`
Memory Fault – Precise Data	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFF`
Memory Fault – STKERR on NMI entry	Priority before NMI was -1 (HardFault).	Lock-up at priority -1 or -2 IMPLEMENTATION DEFINED	`0xFFFFFFFF`

Table B1-8 Lock-up conditions (continued)

Fault cause	Occurrence	Behavior	Lock-up Address
Memory Fault – UNSTKERR	Un-stacking fault	Lock-up at priority -1 or -2 or HardFault exception[a]	`0xFFFFFFFE`
SVC[b]	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFE`
Usage Fault	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFE`
Undefined instruction	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFE`
Breakpoint[c]	Priority -1 or -2	Lock-up at priority of occurrence	`0xFFFFFFFE`

a. The behavior is dependent on the restored state. See *Integrity checks on exception returns* on page B1-22 for more information.
b. At priority -1 or -2, SVC is treated as an UNDEFINED instruction
c. `BKPT` instruction

B1.5.16 Reset management

In ARMv6-M, the Application Interrupt and Reset Control register (*Application Interrupt and Reset Control Register (AIRCR)* on page B3-9) provides a mechanism for a system reset:

- The control bit SYSRESETREQ requests a reset by an external system resource. The system components which are reset by this request are IMPLEMENTATION DEFINED. A Local Reset is required as part of a system reset request.

For SYSRESETREQ, the reset is not guaranteed to take place immediately. A typical code sequence to synchronize reset following a write to the relevant control bit is:

```
      DSB;
Loop  B   Loop;
```

Reset and debug

Debug logic in ARMv6-M is only present when the Debug Extension is implemented and its reset policy is IMPLEMENTATION DEFINED. See *Debug and reset* on page C1-9 for more details.

B1.5.17 Power management

ARMv6-M supports the use of Wait for Interrupt (WFI) and Wait for Event (WFE) instructions as part of a power management policy. Wait for Interrupt provides a mechanism for hardware to support entry to one or more sleep states. Hardware can suspend execution while waiting for a wakeup event. The levels of power saving and associated wakeup latency, while execution is suspended, are IMPLEMENTATION DEFINED.

Wait for Event provides a mechanism for software to suspend program execution with minimal or no impact on wakeup latency until a condition is met. Wait for Event allows some freedom for hardware to instigate power saving measures. Both WFI and WFE are hint instructions and can have no effect. They are generally used in software idle loops that resume program execution after an interrupt or event of interest has occurred.

Note

Code using WFE and WFI must handle spurious wakeup events as a result of a debug halt or other IMPLEMENTATION DEFINED reasons.

For more information, see:

- *Wait For Event and Send Event*
- *Wait For Interrupt* on page B1-35.

Where power management is supported, control and configuration is provided by the System Control Register, see *System Control Register (SCR)* on page B3-10. Support for the following features is provided:

- The transition of an interrupt from the Inactive to the Pending state can be configured as a wakeup event. This allows resumption of a Wait for Event instruction using a masked interrupt as the wakeup event.
- On an exception return, if no exceptions other than the returning exception are Active, there is a configuration bit to suspend execution. When the feature is enabled, the exception return is not performed. The subsequent activation of any exception behaves as a chained exception (see *Tail-chaining* on page B1-27).

  The suspended state can be exited spuriously. ARM recommends that software is written to handle spurious wakeup events and the associated exception return.
- A qualifier that indicates support of different levels of sleep. The bit indicates that the wakeup time from the suspended execution can be longer than if the bit is not set. Typically this can be used to determine whether a PLL or other clock generator can be suspended. The exact behavior is IMPLEMENTATION DEFINED.

B1.5.18 Wait For Event and Send Event

ARMv6-M can support software-based synchronization with respect to system events using the SEV and WFE hint instructions. Software can:

- use the WFE instruction to indicate that it is able to suspend execution of a process or thread until an event occurs, permitting hardware to enter a low power state
- rely on a mechanism that is transparent to software and provides low latency wake up.

The Wait For Event system relies on hardware and software working together to achieve energy saving. For example, stalling execution of a processor until a device or another processor has set a flag:

- the hardware provides the mechanism to enter the Wait For Event low-power state
- software enters a polling loop to determine when the flag is set:
  - the polling processor issues a Wait For Event instruction as part of a polling loop if the flag is clear
  - an event is generated (hardware interrupt or Send Event instruction from another processor) when the flag is set.

The mechanism depends on the interaction of:

- WFE wake-up events, see *WFE wake-up events*
- the Event Register, see *The Event Register*
- the Send Event instruction, see *The Send Event instruction*
- the Wait For Event instruction, see *The Wait For Event instruction* on page B1-35.

WFE wake-up events

The following events are *WFE wake-up events*:

- the execution of an `SEV` instruction on any processor in a multiprocessor system
- any exception entering the Pending state if SEVONPEND in the System Control Register is set
- an asynchronous exception at a priority that pre-empts any currently active exceptions
- a debug event with debug enabled.

The Event Register

The Event Register is a single bit register for each processor in a multiprocessor system. When set, an Event Register indicates that an event has occurred, since the register was last cleared, that might prevent the processor needing to suspend operation on issuing a `WFE` instruction. The following conditions apply to the Event Register:

- The value of the Event Register at reset ~~is UNKNOWN.~~
- The Event Register is set by any `WFE` wake-up event or by the execution of an exception return instruction. For the definition of exception return instructions see *Exception return behavior* on page B1-21.
- The Event Register is only cleared by a `WFE` instruction.
- Software cannot read or write the value of the Event Register directly.

The Send Event instruction

The Send Event instruction causes an event to be signaled to all processors in a multiprocessor system. The mechanism used to signal the event to the processors is IMPLEMENTATION DEFINED. The Send Event instruction generates a wake up event.

The Send Event instruction, `SEV`, is available to both unprivileged and privileged code, see *SEV* on page A6-88.

The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and returns immediately.
- If the Event Register is clear the processor can suspend execution and enter a low-power state. It can remain in that state until the processor detects a WFE wake-up event or a reset. When the processor detects a WFE wake-up event, or earlier if the implementation chooses, the `WFE` instruction completes.

The Wait For Event instruction, `WFE`, is available to both unprivileged and privileged code, see *WFE* on page A6-109.

WFE wake up events can occur before a `WFE` instruction is issued. Software using the Wait For Event mechanism must be tolerant to spurious wake-up events, including multiple wake ups.

Pseudocode details of the Wait For Event lock mechanism

The `SetEventRegister()` pseudocode procedure sets the processor Event Register.

The `ClearEventRegister()` pseudocode procedure clears the processor Event Register.

The `EventRegistered()` pseudocode function returns TRUE if the processor Event Register is set and FALSE if it is clear:

```
boolean EventRegistered()
```

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every processor in a multiprocessor system.

B1.5.19 Wait For Interrupt

In ARMv6-M, Wait For Interrupt is supported through the hint instruction, `WFI`. For more information, see *WFI* on page A6-110.

When a processor issues a `WFI` instruction it can suspend execution and enter a low-power state. It can remain in that state until the processor detects a reset or one of the following *WFI wake-up events*:

- an asynchronous exception at a priority that pre-empts any currently active exceptions

Note

If PRIMASK is set, an asynchronous exception that has a higher group priority than any active exception results in a `WFI` instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.

- a debug event with debug enabled.

When the hardware detects a WFI wake-up event, or earlier if the implementation chooses, the `WFI` instruction completes.

WFI wake-up events are recognized after the `WFI` instruction is issued.

Note

Because debug entry is one of the WFI wake-up events, ARM recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.

Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into power-down routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. forces the suspension of execution, and of all associated bus activity
2. ceases to execute instructions from the processor.

The control logic required to do this typically tracks the activity of the bus interfaces of the processor. This means it can signal to an external power controller that there is no ongoing bus activity.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred power-down entry mechanism for future implementations.

Pseudocode details of Wait For Interrupt

The `WaitForInterrupt()` pseudocode procedure optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.

Chapter B2
System Memory Model

This chapter contains information on the memory model pseudocode, the pseudocode associated with memory accesses. The chapter is made up of the following sections:

B2.1 Introduction

The pseudocode described in this chapter is associated with explicit memory accesses. Implicit accesses can occur due to instruction prefetching in hardware or on exception entry and return.

The pseudocode usage hierarchy is as follows:

- instructions which require a memory access use the helper functions MemA[] or MemU[]
- the access is governed by whether the access is a read or write, its address alignment, data endianness and memory attributes
- memory attributes are determined from the default system address map as defined in *The system address map* on page B3-2.

The pseudocode is broken down into the following subsections:

- declarations and general supporting functions
- memory access support.

The ARMv6-M memory model is compatible with other ARMv6 and ARMv7 architecture variants. Endian configuration and support is described in *Control of the Endian Mapping in ARMv6-M* on page B2-8. To ensure system correctness, barrier instructions are required to provide certain guarantees around accesses to key resources in the System Control Space as described in *Barrier support for system correctness* on page B2-9.

B2.2 Declarations and support functions

For global declarations see *Register related definitions for pseudocode* on page B1-11.
For a list of helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxE-22.

Declarations are as follows:

```
boolean iswrite;       /* TRUE for memory stores, FALSE for load accesses */
boolean ispriv         /* TRUE if the instruction executing with privileged access */
                       /* ispriv is always TRUE in ARMv6-M */
```

General support functions are as follows:

```
// FindPriv()
// ==========

boolean FindPriv()
    if (CurrentMode==Mode_Handler) OR ((CurrentMode==Mode_Thread)AND(CONTROL<0>=='0')) then
        ispriv = TRUE;
    else
        ispriv = FALSE;
    return ispriv;


// IsAligned()
// ===========

boolean IsAligned(bits(32) address, integer size)
    assert size==1 OR size==2 OR size==4;   // for ARMv6-M size must be one of 1,2,4
    mask = (size-1)<31:0>;                  // integer to bit string conversion
    return ((address AND mask) == AllZeroes(32));


// BigEndianReverse()
// ==================

bits(8*N) BigEndianReverse (bits(8*N) value, integer N)
    assert N == 1 || N == 2 || N == 4;
    bits(8*N) result;
    case N of
        when 1
            result<7:0> = value<7:0>;
        when 2
            result<15:8> = value<7:0>;
            result<7:0> =  value<15:8>;
        when 4
            result<31:24> = value<7:0>;
            result<23:16> = value<15:8>;
            result<15:8> = value<23:16>;
            result<7:0> =  value<31:24>;
    return result;
```

B2.2.1 Memory attribute definitions and mapping

The following pseudocode defines the selection of memory attributes with respect to the ARMv6-M address map.

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device, MemType_StronglyOrdered};


// Memory attributes descriptor

type MemoryAttributes is (
    MemType type,
    bits(2) innerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    bits(2) outerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    boolean shareable
)


// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
    bits(32)         physicaladdress
)


// MemoryAccessAddress()
// =====================


// Associate the default address map attributes to the memory access
// ARMv6-M does not qualify memory access with privilege or read/write access - no MPU support


AddressDescriptor MemoryAccessAddress(bits(32) address, boolean ispriv, boolean iswrite)
    AddressDescriptor result;
    result.memattrs = DefaultMemoryAttributes(bits(32) address);
    result.physicaladdress = address;
    return result;


// DefaultMemoryAttributes()
// =========================


MemoryAttributes DefaultMemoryAttributes(bits(32) address, boolean isinstrfetch)
    MemoryAttributes memattrs;

    if (((UInt(PA<31:28>) > 9) OR (PA<31:29> == ’010’)) AND isinstrfetch) then
        ExceptionTaken(HardFault);      // Instruction execution attempted from XN memory

    case PA<31:29> of
```

```
        when ‘000’
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = ‘10’;
            memattrs.shareable = ‘0’;

        when ‘001’
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = ‘01’;
            memattrs.shareable = ‘0’;
        when ‘010’
            memattrs.type = MemType_Device;
            memattrs.innerattrs = ‘00’;
            memattrs.shareable = ‘0’;
        when ‘011’
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = ‘01’;
            memattrs.shareable = ‘0’;
        when ‘100’
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = ‘10’;
            memattrs.shareable = ‘0’;
        when ‘101’
            memattrs.type = MemType_Device;
            memattrs.innerattrs = ‘00’;
            memattrs.shareable = ‘1’;
        when ‘110’
            memattrs.type = MemType_Device;
            memattrs.innerattrs = ‘00’;
            memattrs.shareable = ‘0’;
        when ‘111’
            if PA<28:20> = ‘00000000’ then
                memattrs.type = MemType_StronglyOrdered;
                memattrs.innerattrs = ‘00’;
                memattrs.shareable = ‘1’;
            else
                memattrs.type = MemType_Device;
                memattrs.innerattrs = ‘00’;
                memattrs.shareable = ‘0’;

        // Outer attributes are the same as the inner attributes in all cases.
        memattrs.outerattrs = memattrs.innerattrs;

    return memattrs;
```

B2.3 Memory accesses

The pseudocode used to describe the operation of load and store instructions is defined in terms of unaligned accesses. The underlying structure determines the alignment behavior for cases where an aligned address must be used. ARMv6-M only supports aligned memory accesses that comply to the MemA[] pseudo-function. This is equivalent to supporting aligned accesses only in other architecture variants, for example by:

- setting the CCR_UNALIGN.TRP bit in an ARMv7-M implementation
- setting the SCTLR.A bit in an ARMv7-A or ARMv7-R implementation.

The effect is that the unaligned memory access function MemU[] can be mapped directly to the aligned memory access function MemA[] in ARMv6-M:

```
// For ARMv6-M:

    bits(8*size) MemU[bits(32) address, integer size]
<=> bits(8*size) MemA[bits(32) address, integer size]
```

B2.3.1 The _Mem[] function

The _Mem[] function performs single-copy atomic, aligned, little-endian memory accesses to the underlying physical memory array of bytes:

```
bits(8*size) _Mem[AddressDescriptor memaddrdesc, integer size]          // non-assignment form
    assert size == 1 || size == 2 || size == 4;

_Mem[AddressDescriptor memaddrdesc, integer size] = bits(8*size) value   // assignment form
    assert size == 1 || size == 2 || size == 4;
```

The function addresses the array using a 32-bit physical address supplied in memaddrdesc.physicaladdress.

For ARMv6-M, the 2^{32} byte address space complies with the system address map restrictions, as defined in *The system address map* on page B3-2.

The attributes within memaddrdesc.memattrs are used by the memory system to determine caching and ordering behaviors as described in *Memory types* on page A3-8 and *Access rights* on page A3-15.

B2.3.2 The MemA[] function

The following links provide references to related information for the MemA[] definition:

global declarations	see *Register related definitions for pseudocode* on page B1-11
MemoryAccessAddress()	see *Memory attribute definitions and mapping* on page B2-4
ExceptionTaken()	see *Exception entry behavior* on page B1-18
_Mem[]	see *The _Mem[] function*
other functions	see *Miscellaneous helper procedures and functions* on page AppxE-22

```
// MemA[] non-assignment form used for memory reads
// ================================================

bits(8*size) MemA[bits(32) address, integer size]
    bits(8*size) value;

    // check alignment

    if !IsAligned(address, size) then
        ExceptionTaken(HardFault);
    memaddrdesc = MemoryAccessAddress(address, FindPriv(), FALSE);
    value = _Mem[memaddrdesc, size]  ;
    if AIRCR.ENDIANESS == '1' then
        value = BigEndianReverse(value, size);
    return value;


// MemA[] assignment form used for memory writes
// =============================================

MemA[bits(32) address, integer size] = bits(8*size) value

    // check alignment

    if !IsAligned(address, size) then
        ExceptionTaken(HardFault);
    memaddrdesc = MemoryAccessAddress(address, FindPriv(), TRUE);
    if (memaddrdesc.memattrs.shareable == '1') then
        ClearExclusiveByAddress(memaddrdesc.physicaladdress, ProcessorID(), size);    // see Note
    if AIRCR.ENDIANESS == '1' then
        value = BigEndianReverse(value, size);
    _Mem[memaddrdesc,size] = value;

// Note: ARMv6-M does not support exclusive access. This function activates to clear a global
//       monitor where the fabric and other agents support exclusive accesses, otherwise it is a
//       null function.
```

B2.4 Control of the Endian Mapping in ARMv6-M

ARMv6-M supports a selectable endian model, that is configured to be big endian (BE) or little endian (LE). It is IMPLEMENTATION DEFINED whether the selection is a build time option or determined from a control input on system reset. The endian mapping has the following restrictions:

- The endian setting only applies to data accesses, instruction fetches are always little endian
- Loads and stores to the Private Peripheral Bus (*General rules applying to PPB register access* on page B3-4) are always little endian

The endian configuration can be determined by reading a system control register, see *Application Interrupt and Reset Control Register (AIRCR)* on page B3-9 for details.

Where a big endian instruction format is required with ARMv6-M, byte swapping within a halfword is required in the bus fabric. The byte swap is required for instruction fetches only and must not occur on data accesses.

For example, for instruction fetches over a 32-bit bus:

```
PrefetchInstr<31:24> -> PrefetchInstr<23:16>
PrefetchInstr<23:16> -> PrefetchInstr<31:24>
PrefetchInstr<15:8> -> PrefetchInstr<7:0>
PrefetchInstr<7:0> -> PrefetchInstr<15:8>
```

B2.5 Barrier support for system correctness

ARMv6-M has limited support for ensuring that system configuration events have completed as part of system correctness. For special-purpose register support, see *The special-purpose control register* on page B1-10 and *Special-purpose register updates and the memory order model* on page B1-11.

All other resources are memory mapped and it is important for system correctness that exception management software can ensure:

- SVCall and PendSV exception priorities can be changed ~~where the OS Extension is supported~~ while ~~there~~ are not in the Active state.
- SysTick (~~OS Extension only)~~ and NVIC generated interrupts can be disabled and it is deterministic when the associated exception will not enter the Active state.

To meet the condition for SVCall or PendSV, priority management should be done in Thread mode, otherwise software is required to ensure neither exception can transition to the active state while SHPR3 (see *System Handler Priority Register 3 (SHPR3)* on page B3-11) is updated.

For ARMv6-M, the following context guarantees between memory-mapped registers and barrier instructions are required:

- Stores to the System Control Space (SCS) must be followed by a `DSB` instruction to ensure execution of the system control operation.
- An `ISB` instruction must ensure all SCS system control operations in program order before the `ISB` instruction are complete and all side-effects are visible to all instructions in program order fetched after the `ISB` instruction.

The following sequence is required to ensure registers within the System Control Space (see Table B3-2 on page B3-5) are updated:

```
SCS_RegisterWrite();    // write access to the System Control Space register
DSB;                    // ensure the write occurs to the register
ISB;                    //ensure any context guarantees have completed
```

The following are examples of where the barrier support provides the required correctness:

- For SysTick support ~~in the OS Extension,~~ a write to clear the STCSR.ENABLE bit including the barrier instruction sequence ensures that the ICSR.PENDSTSET bit will not change from the Inactive to Pending state. A subsequent write to set ICSR.PENDSTCLR including the barrier instruction sequence ensures that activation of the SysTick exception is prevented beyond this point.
- A write of an ICER.CLRENA bit to disable an external interrupt source including the barrier instruction sequence ensures that the associated ISPR.SETPEND bit will not become active beyond this point.
- ICSR.VECTPENDING will not change state due to a SysTick or NVIC interrupt after the associated exception disable sequences described above are executed.

Chapter B3
System Address Map

This chapter contains information on the system address map. It contains the following sections:

B3.1 The system address map

For ARMv6-M, the 32-bit address space is predefined, with subdivision for code, data, and peripherals, as well as regions for on-chip (tightly coupled to the core) and off-chip resources. The address space supports 8 x 0.5GB primary partitions:

- Code
- SRAM
- Peripheral
- 2 x RAM regions
- 2 x Device regions
- System

Physical addresses are architecturally assigned for use as event entry points (vectors), system control, and configuration. The event entry points are all with respect to a table base address, ~~where the base address is automatically set to `0x00000000` on reset, then maintained in an address space reserved for system configuration and control. To meet this and other system needs, the~~ address space `0xE0000000` to `0xFFFFFFFF` is reserved for system level use.

Table B3-1 on page B3-3 describes the ARMv6-M address map.

- XN refers to eXecute Never for the region and will fault (~~MemManage~~ exception) any attempt to execute in the region.
- The Cache column indicates inner/outer cache policy to support system caches. The policy allows a declared cache type to be demoted but not promoted.

  WT: write through, can be treated as non-cached

  WBWA: write-back, write allocate, can be treated as write-through or non-cached
- Shareable indicates to the system that the access is intended to support shared use from multiple agents; multiple processors and/or DMA agents within a coherent memory domain.

  Note

  ARMv6-M does not support exclusive access instructions (`LDREX{B,H}`, `STREX{B,H}`) or any form of atomic swap instruction. Software needs to take account of this in multiprocessing environments which use shared memory.
- It is IMPLEMENTATION DEFINED which portions of the overall address space are designated read-write, which are read-only (for example Flash memory), and which are no-access (unpopulated parts of the address map).
- A multi-word access which crosses a 0.5GB address boundary is UNPREDICTABLE.

For additional information on memory attributes and the memory model see Chapter A2 *Application Level Programmers' Model*.

Table B3-1 ARMv6-M address map

	Name	Device Type	XN	Cache	Description

`0x00000000-`
`0x1FFFFFFF`
	Code	Normal	-	WT	Typically ROM or flash memory. Memory required from address 0x0 to support the vector table for system boot code on reset.

`0x20000000-`
`0x3FFFFFFF`
	SRAM	Normal	-	WBWA	SRAM region typically used for on-chip RAM.

`0x40000000-`
`0x5FFFFFFF`
	Peripheral	Device	XN	-	on-chip peripheral address space

`0x60000000-`
`0x7FFFFFFF`
	RAM	Normal	-	WBWA	memory with write-back, write allocate cache attribute for L2/L3 cache support

`0x80000000-`
`0x9FFFFFFF`
	RAM	Normal	-	WT	memory with write-through cache attribute

`0xA0000000-`
`0xBFFFFFFF`
	Device	Device, Shareable	XN	-	shareable device space

`0xC0000000-`
`0xDFFFFFFF`
	Device	Device	XN	-	non-shareable device space

`0xE0000000-`
`0xFFFFFFFF`
	System	-	-	-	system segment including the PPB

+0000000	PPB	SO, (Shareable)	XN	-	1MB region reserved as a Private Peripheral Bus. The PPB supports key resources including the System Control Space, and debug features.
+0100000	Vendor_SYS	-	-	-	Reserved in ARMv6-M

B3.1.1 General rules applying to PPB register access

The Private Peripheral Bus (PPB), address range `0xE0000000` to `0xE0100000`, supports the following general rules:

- The region is defined as Strongly Ordered memory – see *Strongly Ordered memory attribute* on page A3-13 and *Memory access restrictions* on page A3-13.
- Registers are always accessed little endian regardless of the endian state of the processor.
- The PPB address space only supports (aligned) word accesses. Byte and halfword access is UNPREDICTABLE.

  Note

  This is different from ARMv7-M where byte and halfword accesses are supported in several cases. For ARMv6-M, software must perform read-modify-write accesses sequences where it needs to modify byte fields within a word in the PPB memory region.

- The term set means writing the value to 1, and the term clear(ed) means writing the value to 0. Where the term applies to multiple bits, all bits assume the written value.
- The term disable means writing the bit value to 0, the term enable means writing the bit value to 1.
- Where a bit is defined as clear on read, the following atomic behavior must be guaranteed when the bit is being read coincident with an event which sets the bit
  - — If the bit reads as one, the bit is cleared by the read operation
  - — If the bit reads as zero, the bit is set and read/cleared by a subsequent read operation
- A reserved register or bit field must be treated as UNK/SBZP.

For debug related resources, see *General rules applying to debug register access* on page C1-5.

B3.2 System Control Space (SCS)

The System Control Space is a memory-mapped 4kB address space which is used along with the special-purpose registers to provide arrays of 32-bit registers for configuration, status reporting and control. The ARMv6-M SCS breaks down into the following groups:

- CPUID space.
- System control, configuration and status.
- A SysTick system timer (~~OS Extension only)~~.
- A Nested Vectored Interrupt Controller (NVIC), supporting up to 32 discrete external interrupts. The NVIC shares a common prioritization model with the other exceptions. Additional register space is reserved.
- System debug – see Chapter C1 *ARMv6-M Debug*.

Table B3-2 defines the address space breakdown of the SCS register groups.

Table B3-2 SCS address space regions

System Control Space (address range `0xE000E000` to `0xE000EFFF`)

Group	Address Range(s)	Notes
System Control/ID	`0xE000E000-0xE000E00F`	includes the Auxiliary Control register
	`0xE000ED00-0xE000ED8F`	System control and ID registers
	`0xE000EF90-0xE000EFCF`	IMPLEMENTATION DEFINED
SysTick	`0xE000E010-0xE000E0FF`	System Timer (~~OS Extension only)~~
NVIC	`0xE000E100-0xE000ECFF`	External interrupt controller
Debug	`0xE000EDF0-0xE000EEFF`	Debug control and configuration (Debug Extension only)
	NOTE: unassigned addresses are reserved	

Detailed breakdown of the register groups is provided as part of the appropriate feature definition:

- System ID block in *System ID register support in the SCS* on page B3-7
- System control and configuration in *The System Control Block (SCB)* on page B3-7
- SysTick system timer in *System timer - SysTick* on page B3-13
- NVIC in *Nested Vectored Interrupt Controller (NVIC)* on page B3-17
- Debug in Chapter C1 *ARMv6-M Debug*

B3.2.1 System control and ID blocks

System control and ID is supported by registers within subregions of the System Control Space as defined in Table B3-3.

Table B3-3 ARMv6-M System control and ID registers

Address	Type	Reset Value	Name	Function
`0xE000E008`	Read/Write	IMPLEMENTATION DEFINED	ACTLR	Auxiliary Control Register
...				...
`0xE000ED00`	Read Only	IMPLEMENTATION DEFINED	CPUID	CPUID Base Register
`0xE000ED04`	Read/Write	`0x00000000`	ICSR	Interrupt Control State Register
`0xE000ED0C`	Read/Write	bits [10:8] = ’000’	AIRCR	Application Interrupt/Reset Control Register
`0xE000ED10`	Read/Write	bits [4,2,1] = ’000’	SCR	System Control Register (optional)
`0xE000ED14`	Read Only	bits [9,3] = ’1’	CCR	Configuration and Control Register
`0xE000ED1C`	Read/Write	`SBZ`[a]	SHPR2	~~OS Extension, otherwise reserved~~
`0xE000ED20`	Read/Write	`SBZ`[b]	SHPR3	~~OS Extension, otherwise reserved~~
`0xE000ED24`	Read/Write	`0x00000000`	SHCSR	System Handler Control and State Register
`0xE000ED30`	Read/Write	`0x00000000`	DFSR	DebugFault Status Register (Debug Extension only)
`0xE000EDF0` - `0xE000EEFF`				See *Debug register support in the SCS* on page C1-12

a. Bits [31:30] are zero ~~where the OS Extension is implemented~~ (SVCall priority).

b. Bits [31:30] and bits [23:22] are zero ~~where the OS Extension is implemented~~ (SysTick and PendSV priorities).

The Auxiliary Control Register (ACTLR)

The Auxiliary Control Register, ACTLR, provides implementation-specific configuration and control options. The contents of this register are IMPLEMENTATION DEFINED.

Table B3-4 Auxiliary Control Register – (0xE000E008)

Bits	R/W	Name	Function
[31:0]			IMPLEMENTATION DEFINED

B3.2.2 System ID register support in the SCS

Support consists of the CPUID base register described in Table B3-5 and a block of ID attribute registers.

Table B3-5 CPUID Base Register – (CPUID, 0xE000ED00)

Bits	R/W	Name	Function
[31:24]	RO	IMPLEMENTER	Implementer code assigned by ARM. ARM == `0x41`.
[23:20]	RO	VARIANT	IMPLEMENTATION DEFINED
[19:16]	RO	(Constant)	Reads as `0xC` for ARMv6-M parts
[15:4]	RO	PARTNO	IMPLEMENTATION DEFINED
[3:0]	RO	REVISION	IMPLEMENTATION DEFINED

The CPUID allows applications and debuggers to determine the processor is compliant to ARMv6-M. It is IMPLEMENTATION DEFINED whether the information defines the presence of the OS and/or Debug Extensions. This register is word accessible only.

B3.2.3 The System Control Block (SCB)

Key control and status features of ARMv6-M are managed centrally in a System Control Block within the SCS. The SCB provides support for the following features:

- Software reset control at various levels
- Base address management (table pointer control) for the exception model
- System exception management (excludes external interrupts handled by the NVIC)
  - Exception enables
  - Setting or clearing exceptions to/from the pending state
  - Exception status (Inactive, Pending, or Active). Inactive is when an exception is neither Pending nor Active.
  - Priority setting (for configurable system exceptions)
  - Miscellaneous control and status information
- The exception (vector) number of the currently executing code and highest pending exception
- Miscellaneous control and status features
- Debug status information – supplemented with control and status in the debug specific register region. See Chapter C1 *ARMv6-M Debug* for debug details.

The SCB registers are listed in the following subsections in the order of their access address.

Interrupt Control State Register (ICSR)

Table B3-6 Interrupt Control and State Register – (0xE000ED04)

Bits	R/W	Name	Function
[31]	R/W	NMIPENDSET	Setting this bit will activate an NMI. Since NMI is the highest priority exception, it will activate as soon as it is registered. Reads back with current state (1 if Pending, 0 if not).
[28]	R/W	PENDSVSET[a]	Set a pending PendSV interrupt. This is normally used to request a context switch. Reads back with current state (1 if Pending, 0 if not). ~~OS Extension only, otherwise RAZ/WI.~~
[27]	WO	PENDSVCLR[a]	Clear a pending PendSV interrupt. ~~OS Extension only, otherwise reserved.~~
[26]	R/W	PENDSTSET[b]	Set a pending SysTick. Reads back with current state (1 if Pending, 0 if not). ~~OS Extension only, otherwise~~ RAZ/WI.
[25]	WO	PENDSTCLR[b]	Clear a pending SysTick (whether set here or by the timer hardware). ~~OS Extension only, otherwise reserved.~~
[23]	RO	ISRPREEMPT	If set, a pending exception will be serviced on exit from the debug halt state. The bit applies to the Debug Extension only, otherwise it is reserved.
[22]	RO	ISRPENDING	Indicates if an external configurable (NVIC generated) interrupt is pending. This bit applies to the Debug Extension only, otherwise it is reserved.
[20:12]	RO	VECTPENDING	Indicates the exception number for the highest priority pending exception. The pending state includes the effect of memory-mapped enable and mask registers. It does not include the PRIMASK special-purpose register qualifier. A value of zero indicates no pending exceptions. ~~This field applies to the OS Extension only, otherwise it is reserved.~~
[8:0]	RO	VECTACTIVE	0: Thread mode value > 1: the exception number[c] for the current executing exception. Debug Extension only, otherwise reserved.
Unused			Reserved

a. writing PENDSVSET and PENDSVCLR to '1' concurrently is UNPREDICTABLE
b. writing PENDSTSET and PENDSTCLR to '1' concurrently is UNPREDICTABLE
c. this is the same value as IPSR<8:0>

Vector Table Offset Register (VTOR)

The vector table base address is fixed at 0x00000000. This register (address 0xE000ED08) is RAZ/WI for ARMv6-M.

Application Interrupt and Reset Control Register (AIRCR)

AIRCR.ENDIANESS can be used to determine the system endianness for data access, see *Endian support* on page A3-4for more details.

AIRCR.SYSRESETREQ is required to generate a Local Reset. see *Reset management* on page B1-32 for details.

Table B3-7 Application Interrupt and Reset Control Register – (0xE000ED0C)

Bits	R/W	Name	Function
[31:16]	WO	VECTKEY	Vector Key. Should be written with the value `0x05FA`, otherwise the register write is UNPREDICTABLE.
[31:16]	RO	VECTKEYSTAT	UNPREDICTABLE
[15]	RO	ENDIANESS[a]	value == 0: little endian (LE) value == 1: big endian (BE)
[2]	WO	SYSRESETREQ	Writing this bit 1 will cause a signal to be asserted to the external system to indicate a reset is requested. The bit self-clears as part of the reset sequence.
[1]	WO	VECTCLRACTIVE[b]	Clears all active state information for fixed and configurable exceptions. This bit self-clears and is classified as a debug resource, see Table C1-1 on page C1-2. The bit can only be written when the core is halted. Note: It is the debugger's responsibility to re-initialize the stack.
Unused			Reserved

a. It is IMPLEMENTATION DEFINED whether the bit is a build option or fixed on reset, see *Control of the Endian Mapping in ARMv6-M* on page B2-8.

b. It is IMPLEMENTATION DEFINED whether VECTCLRACTIVE also clears pending status associated with any non-configurable exceptions, specifically HardFault and NMI.

System Control Register (SCR)

The System Control Register is required when power management support is implemented in the `WFI` and `WFE` hint instructions:

- For more information on WFE, see *WFE* on page A6-109
- For more information on WFI, see *WFI* on page A6-110.

Table B3-8 System Control Register (0xE000ED10)

Bits	R/W	Name	Function
[4]	R/W	SEVONPEND	When enabled, interrupt transitions from Inactive to Pending are included in the list of wakeup events for the `WFE` instruction. See *WFE wake-up events* on page B1-34 for more information.
[2]	R/W	SLEEPDEEP	A qualifying hint that indicates waking from sleep might take longer. Implementations can take advantage of the feature to identify a lower power sleep state.
[1]	R/W	SLEEPONEXIT	~~When set, the implementation can enter a sleep state on an exception return that loads the IPSR with ExceptionNumber == 0.~~
Unused			Reserved

A debugger can read S_SLEEP (see *Debug Halting Control and Status Register (DHCSR)* on page C1-13) to detect if sleeping.

Configuration and Control Register (CCR)

Table B3-9 Configuration and Control Register – (0xE000ED14)

Bits	R/W	Name	Function
[9]	RO	STKALIGN	RAO: on exception entry, the SP used prior to the exception is adjusted to be 8-byte aligned and the context to restore it is saved. The SP is restored on the associated exception return.
[3]	RO	UNALIGN_TRP	RAO: unaligned word and halfword accesses generate a HardFault exception.
Unused			Reserved

System Handler Priority Register 2 (SHPR2)

Table B3-10 System Handler Priority Register 2 – (0xE000ED1C)

Bits	R/W	Name	Function
[31:30]	R/W	PRI_11	Priority of system handler 11 - SVCall
[29:0]			Reserved

System Handler Priority Register 3 (SHPR3)

Table B3-11 System Handler Priority Register 3 – (0xE000ED20)

Bits	R/W	Name	Function
[31:30]	R/W	PRI_15	Priority of system handler 15 - SysTick
[29:24]	R/W		Reserved
[23:22]	R/W	PRI_14	Priority of system handler 14 - PendSV
[21:0]	R/W		Reserved

System Handler Control and State Register (SHCSR)

Table B3-12 System Handler Control and State Register – (0xE000ED24)

Bits	R/W	Name	Function
[31:16]			Reserved
[15]	R/W	SVCALLPENDED	Reads as 1 if SVCall is Pending[a]
[14:0]			Reserved

a. A state bit that is set when the exception started to invoke but was replaced by a higher priority exception.

Note

For ARMv6-M, this register is only accessible as part of the debug extension (via the DAP port, see *The Debug Access Port (DAP)* on page C1-4), and not through the system memory map as viewed by the core.

Fault Status Registers

ARMv6-M only supports fault status information as part of the Debug Extension.

Debug Fault Status Register (DFSR)

This register is part of the Debug Extension. See Chapter C1 *ARMv6-M Debug* for a full description of debug support within ARMv6-M and *Debug Fault Status Register (DFSR)* on page C1-12 for the Debug Fault Status Register.

B3.3 System timer - SysTick

ARMv6-M supports a system timer, SysTick, ~~in the OS Extension.~~ SysTick provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism. The counter can be used in several different ways, for example:

- An RTOS tick timer which fires at a programmable rate (for example 100Hz) and invokes a SysTick routine.
- A high speed alarm timer using Core clock.
- A variable rate alarm or signal timer – the duration range dependent on the reference clock used and the dynamic range of the counter.
- A simple counter. Software can use this to measure time to completion and time used.
- An internal clock source control based on missing/meeting durations. The COUNTFLAG bit-field in the control and status register can be used to determine if an action completed within a set duration, as part of a dynamic clock management control loop.

B3.3.1 Theory of operation

The timer consists of four registers:

- A control and status counter to configure its clock, enable the counter, enable the SysTick interrupt, and determine counter status.
- The reload value for the counter, used to provide the counter's wrap value.
- The current value of the counter.
- A calibration value register, indicating the preload value necessary for a 10ms (100Hz) system clock.

When enabled, the timer will count down from ~~the reload value to zero, reload (wrap) to~~ the value in the SysTick ~~Reload~~ Value Register ~~on the next clock edge, then decrement on subsequent clocks. Writing a value of zero to the Reload Value Register disables the counter on the next wrap~~. When the counter transitions to zero, the COUNTFLAG status bit is set. The COUNTFLAG bit clears on reads.

~~Writing to the Current Value Register will clear the register and the COUNTFLAG status bit. The write does not trigger the SysTick exception logic. On a read, the current value is the value of the register at the time the register is accessed.~~

The calibration value TENMS allows software to scale the counter to other desired clock rates within the counter's dynamic range.

If the core is in Debug state (halted), the counter will not decrement.

The timer is clocked with respect to a reference clock. The reference clock can be the core clock or an external clock source. Where an external clock source is used, the implementation must document the relationship between the core clock and the external reference. This is required for system timing calibration, taking account of metastability, clock skew and jitter.

B3.3.2 System timer register support in the SCS

Table B3-13 summarizes the register support provided within the SCS address map. The SysTick registers ~~are only present if the OS Extension is implemented, otherwise~~ the registers are reserved.

Table B3-13 SysTick register support in the SCS

Address	R/W	Reset Value	Name	Function
`0xE000E010`	R/W	0x00000000	SYST_CSR	SysTick Control and Status
`0xE000E014`	R/W	UNKNOWN	SYST_RVR	SysTick Reload value
`0xE000E018`	R/W	UNKNOWN	SYST_CVR	SysTick Current value
`0xE000E01C`	RO	IMP DEF	SYST_CALIB	SysTick Calibration value
…to `0xE000E0FF`				Reserved

Descriptions of the four SysTick registers are provided in the following subsections.

SysTick Control and Status Register (SYST_CSR)

Table B3-14 SysTick Control and Status Register – (0xE000E010)

Bits	R/W	Name	Function
[16]	RO	COUNTFLAG	Returns 1 if timer counted to 0 since last time this register was read. COUNTFLAG is set by a count transition from 1 => 0. COUNTFLAG is cleared on read or by a write to the Current Value register.
[2]	R/W	CLKSOURCE	0: clock source is (optional) external reference clock 1: core clock used for SysTick If no external clock provided, this bit will read as 1 and ignore writes.
[1]	R/W	TICKINT	1: counting down to 0 will cause the SysTick exception to be pended. Clearing the SysTick Current Value register by a register write in software will not cause SysTick to be pended. 0: counting down to 0 does not cause the SysTick exception to be pended. Software can use COUNTFLAG to determine if a count to zero has occurred.
[0]	R/W	ENABLE	0: the counter is disabled 1: the counter will operate in a multi-shot manner.
Unused			Reserved

SysTick Reload Value Register (SYST_RVR)

Table B3-15 SysTick Reload Value Register – (0xE000E014)

Bits	R/W	Name	Function
[31:24]	RAZ/WI		
[23:0]	R/W	RELOAD	Value to load into the Current Value register when the counter reaches 0.

SysTick Current Value Register (SYST_CVR)

Table B3-16 SysTick Current Value Register – (0xE000E018)

Bits	R/W	Name	Function
[31:0]	R/W	CURRENT	Current counter value. This is the value of the counter at the time it is sampled. The counter does not provide read-modify-write protection. The register is write-clear. A software write of any value will clear the register to 0. Unsupported bits RAZ (see SysTick Reload Value register).

SysTick Calibration value Register (SYST_CALIB)

Table B3-17 SysTick Calibration Value Register – (0xE000E01C)

Bits	R/W	Name	Function
[31]	RO	NOREF	If reads as 1, the Reference clock is not provided – the CLKSOURCE bit of the SysTick Control and Status register will be forced to 1 and cannot be cleared to 0.
[30]	RO	SKEW	If reads as 1, the calibration value for 10ms is inexact (due to clock frequency).
[29:24]			Reserved
[23:0]	RO	TENMS	An optional Reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as 0, the calibration value is not known.

B3.4 Nested Vectored Interrupt Controller (NVIC)

ARMv6-M provides an interrupt controller as an integral part of the exception model. The interrupt controller operation aligns with ARM's General Interrupt Controller (GIC) specification, promoted for use with other architecture variants and ARMv7 profiles.

The ARMv6-M NVIC architecture supports up to 32 (IRQ[31:0]) discrete interrupts. The general registers associated with the NVIC are all accessible from a block of memory in the System Control Space as described in Table B3-18 on page B3-19.

B3.4.1 Theory of operation

ARMv6-M supports level-sensitive and pulse-sensitive (a variant of an edge sensitive) interrupt behavior. This means that both level-sensitive and pulse-sensitive interrupts can be handled. Pulse interrupt sources must be held long enough to be sampled reliably by the core clock to ensure they are latched and become Pending. A subsequent pulse can become Pending while the interrupt is Active, however, multiple pulses which occur during the Active period will only register as a single event for interrupt scheduling.

In summary:

- Pulses held for a clock period will act like edge-sensitive interrupts. These can re-pend while the interrupt is Active.

  Note

  A pulse must be cleared before the assertion of AIRCR.VECTCLRACTIVE or the associated exception return, otherwise the interrupt signal behaves as a level-sensitive input and the pending bit is asserted again.

- Level-sensitive interrupts become Pending on assertion and activate according to the exception priority rules. A level-sensitive interrupt input must remain asserted until it is cleared by an appropriate access to the peripheral by the Interrupt Service Routine (ISR). If the interrupt is still asserted on return from the interrupt, it will become Pending again.

All NVIC interrupts have a programmable priority value and an associated exception number as part of the ARMv6-M exception model and its prioritization policy.

The NVIC supports the following features:

- NVIC interrupts can be enabled and disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit-field. The registers use a write-1-to-enable and write-1-to-clear policy, both registers reading back the current enabled state of the corresponding (32) interrupts.

  When an interrupt is disabled, interrupt assertion will cause the interrupt to become Pending, however, the interrupt will not activate. If an interrupt is Active when it is disabled, it remains in its Active state until cleared by reset or an exception return. Clearing the enable bit prevents new activations of the associated interrupt.

  Interrupt enable bits can be hard-wired to zero where the associated interrupt line does not exist, or hard-wired to one where the associated interrupt line cannot be disabled.

- NVIC interrupts can be pended/un-pended using a complementary pair of registers to those used to enable/disable the interrupts, named the Set-Pending Register and Clear-Pending Register respectively. The registers use a write-1-to-enable and write-1-to-clear policy, both registers reading back the current pended state of the corresponding (32) interrupts. The Clear-Pending Register has no effect on the execution status of an Active interrupt.

  It is IMPLEMENTATION DEFINED for each interrupt line supported, whether an interrupt supports setting and/or clearing of the associated pend bit under software control.

- ~~Active bit status is provided to allow software to determine whether an interrupt is Inactive, Active, Pending, or Active and Pending.~~

- NVIC interrupts are prioritized by updating an 8-bit field within a 32-bit register (each register supporting four interrupts). Priorities are maintained according to the ARMv6-M prioritization scheme. See *Exception priorities and pre-emption* on page B1-15.

External interrupt input behavior

The following pseudocode describes the relationship between external interrupt inputs and the NVIC behavior:

```
// DEFINITIONS

NVIC[] is an array of active high external interrupt input signals;
    // the type of signal (level or pulse) and its assertion level/sense is IMPLEMENTATION DEFINED
    // and might not be the same for all inputs

boolean Edge(integer INTNUM);    // Returns true if on a clock edge NVIC[INTNUM]
                                 // has changed from '0' to '1'
boolean NVIC_Pending[INTNUM];    // an array of pending status bits for the external interrupts
integer INTNUM;                  // the external interrupt number


    // The WriteToRegField helper function returns TRUE on a write of '1' event
    // to the field FieldNumber of the RegName register.

boolean WriteToRegField(register RegName, integer FieldNumber)


boolean ExceptionIN(integer INTNUM);   // returns TRUE if exception entry in progress
                                       // to activate INTNUM
boolean ExceptionOUT(integer INTNUM);  // returns TRUE if exception return in progress
                                       // from active INTNUM


// INTERRUPT INTERFACE

sampleInterruptHi = WriteToRegField(AIRCR, VECTCLRACTIVE) || ExceptionOUT(INTNUM);
sampleInterruptLo = WriteToRegField(ICPR, INTNUM);

InterruptAssertion   =  Edge(INTNUM) || (NVIC[INTNUM] && sampleInterruptHi);
InterruptDeassertion =  !NVIC[INTNUM] && sampleInterruptLo;
```

```
// NVIC BEHAVIOR

clearPend = ExceptionIN(INTNUM) || InterruptDeassertion;
setPend   = InterruptAssertion || WriteToRegField(ISPR, INTNUM);

if clearPend && setPend then
    IMPLEMENTATION DEFINED whether NVIC_Pending[INTNUM] is TRUE or FALSE;
else
    NVIC_Pending[INTNUM] = setPend || (NVIC_Pending[INTNUM] && !clearPend);
```

B3.4.2 NVIC register support in the SCS

The system control region includes status and configuration registers which apply to the NVIC as part of the general exception model.

All other external interrupt specific registers reside within the NVIC region of the SCS. Table B3-18 summarizes the NVIC specific registers in the SCS.

Table B3-18 NVIC register support in the SCS

Address	Type	Reset Value	Name	Function
0xE000E100	R/W	0x00000000	NVIC_ISER	Irq 0 to 31 Set-Enable Register
0xE000E180	R/W	0x00000000	NVIC_ICER	Irq 0 to 31 Clear-Enable Register
0xE000E200	R/W	0x00000000	NVIC_ISPR	Irq 0 to 31 Set-Pending Register
0xE000E280	R/W	0x00000000	NVIC_ICPR	Irq 0 to 31 Clear-Pending Register
0xE000E400	R/W	0x00000000	NVIC_IPR0	Irq 0 to 3 Priority Register
…	…	…		…
0xE000E41C	R/W	0x00000000	NVIC_IPR7	Irq 28 to 31 Priority Register
unused				Reserved

Interrupt Set-Enable and Clear-Enable Registers (NVIC_ISER and NVIC_ICER)

Table B3-19 Interrupt Set-Enable Register – (0xE000E100)

Bits	R/W	Name	Function
[31:0]	R/W	SETENA	Enable one or more interrupts within a group of 32. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc.). Writing a 1 will enable the associated interrupt. Writing a 0 has no effect. The register reads back with the current enable state.

Table B3-20 Interrupt Clear-Enable Register – (0xE000E180)

Bits	R/W	Name	Function
[31:0]	R/W	CLRENA	Disable one or more interrupts within a group of 32. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc.). Writing a 1 will disable the associated interrupt. Writing a 0 has no effect. The register reads back with the current enable state.

Interrupt Set-Pending and Clear-Pending Registers (NVIC_ISPR and NVIC_ICPR)

The current pending state read from these registers reflects the current status of the hardware inputs and software writes to the registers. It is not affected by the Interrupt Set-Enable Register or Interrupt Clear-Enable Register.

Table B3-21 Interrupt Set-Pending Register – (0xE000E200)

Bits	R/W	Name	Function
[31:0]	R/W	SETPEND	Writing a 1 to a bit pends the associated interrupt under software control. Each bit represents an interrupt number from 0 to 31. Writing a 0 to a bit has no effect on the associated interrupt. The register reads back with the current pending state.

Table B3-22 Interrupt Clear-Pending Register – (0xE000E280)

Bits	R/W	Name	Function
[31:0]	R/W	CLRPEND	Writing a 1 to a bit un-pends the associated interrupt under software control. Each bit represents an interrupt number from 0 to 31. Writing a 0 to a bit has no effect on the associated interrupt. The register reads back with the current pending state.

Interrupt Priority Register (NVIC_IPRx)

Table B3-23 Interrupt Priority Registers – (0xE000E400-E41C)

Bits	R/W	Name	Function
[31:30]	R/W	PRI_N3	Priority of interrupt number N+3 (3, 7, 11, etc.).
[23:22]	R/W	PRI_N2	Priority of interrupt number N+2 (2, 6, 10, etc.).
[15:14]	R/W	PRI_N1	Priority of interrupt number N+1 (1, 5, 9, etc.).
[7:6]	R/W	PRI_N	Priority of interrupt number N (0, 4, 8, etc.).
unused			Reserved

Chapter B4
ARMv6-M System Instructions

As previously stated, ARMv6-M only executes instructions in Thumb state. The full list of supported instructions is provided in *Alphabetical list of ARMv6-M Thumb instructions* on page A6-14. System instructions support reading and writing the special-purpose registers under software control.

B4.1 Alphabetical list of ARMv6-M system instructions

The ARMv6-M system instructions are defined in this section:

- *CPS* on page B4-3
- *MRS* on page B4-5
- *MSR (register)* on page B4-8

Note

MSR(immediate) is a valid instruction in other ARM architecture variants. The MSR(immediate) encoding is UNDEFINED in ARMv6-M.

B4.1.1 CPS

Change Processor State changes the PRIMASK special-purpose register value.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`CPS<effect> <iflags>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	(0)

```
enable = (im == ‘0’);  disable = (im == ‘1’);  changemode = FALSE;
affectPRI = (I == ‘1’);
if InITBlock() then UNPREDICTABLE;
```

Assembler syntax

`CPS<effect><q> i`

where:

`<effect>` Specifies the effect required on PRIMASK. This is one of:

- `IE` Interrupt Enable. This sets the specified bits to 0.
- `ID` Interrupt Disable. This sets the specified bits to 1.

`<q>` See *Standard assembler syntax fields* on page A6-7.

`i` Indicates that PRIMASK is affected. Raises the current priority to 0 when set to 1. PRIMASK is a 1-bit register, which supports privileged access only.

Operation

```
EncodingSpecificOperations();
if enable then
    if affectPRI then PRIMASK<0> = '0';
if disable then
    if affectPRI then PRIMASK<0> = '1';
```

Exceptions

None.

Notes

Privilege Always available in ARMv6-M.

Masks and CPS

The `CPSIE` and `CPSID` instructions are equivalent to using an `MSR` instruction:

- The `CPSIE i` instruction is equivalent to writing a 0 into PRIMASK
- The `CPSID i` instruction is equivalent to writing a 1 into PRIMASK

B4.1.2 MRS

Move to Register from Special Register moves the value from the selected special-purpose register into a general-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`MRS<c> <Rd>,<spec_reg>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				SYSm							

```
d = UInt(Rd);
if d IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16~~..~~20}) then UNPREDICTABLE;
```

Assembler syntax

```
MRS<c><q>  <Rd>, <spec_reg>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	Specifies the destination register.
`<spec_reg>`	Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer ~~(OS Extension only, otherwise UNPREDICTABLE)~~	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
CONTROL	The special-purpose control register	20 [b]
RSVD	Reserved	unused

a. Raises the current priority to 0 when set to 1. This is a 1-bit register.
b. The control register is composed of the following bits:
[0] = RAZ in ARMv6-M. Thread mode is always privileged.
[1] = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack
(PSP if Thread mode ~~and OS Extension implemented,~~ reserved if Handler mode).
This bit resets to 0.

Operation

```
R[d] = 0;
case  SYSm<7:3> of
    when '00000'
        if SYSm<0> == '1' then
            R[d]<8:0> = IPSR<8:0>;
        if SYSm<1> == '1' then
            R[d]<24> = '0';           // T-bit reads as zero
        if SYSm<2> == '0' then
            R[d]<31:27> = APSR<31:27>;
    when '00001'
        case SYSm<2:0> of
            when '000'
                R[d] = SP_main;
            when '001'
                R[d] = SP_process;
    when '00010'
        case SYSm<2:0> of
            when '000'
                R[d]<0> = PRIMASK<0>;
            when `100`
                R[d]<1> = CONTROL<1>;
```

Exceptions

None.

Notes

Privilege	ARMv6-M always executes as privileged.
EPSR	None of the EPSR bits are readable during normal execution. They all read as 0 when read using MRS (Halting debug can read them via the register transfer mechanism).
Bit positions	The PSR bit positions are defined in *The special-purpose program status registers (xPSR)* on page B1-8.

B4.1.3 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose register to the selected special-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`MSR<c> <spec_reg>,<Rn>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn				1	0	(0)	0	(1)	(0)	(0)	(0)	SYSm							

```
n = UInt(Rn);
if n IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16~~..~~20}) then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	Is the general-purpose register to receive the special register contents.
`<spec_reg>`	Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register ~~(reads as zero, see Notes)~~	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer ~~(OS Extension only, otherwise UNPREDICTABLE)~~	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
CONTROL	The special-purpose control register	20 [b]
RSVD	Reserved	unused

a. Raises the current priority to 0 when set to 1. This is a 1-bit register.
b. The control register is composed of the following bits:
<0> = RAZ in ARMv6-M. Thread mode is always privileged.
<1> = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack
(PSP if Thread mode ~~and OS Extension implemented,~~ reserved if Handler mode).
This bit resets to 0.

Operation

```
case  SYSm<7:3> of
    when '00000'
        if SYSm<2> == '0' then
            APSR = R[n]<31:27>;
    when '00001'
        case SYSm<2:0> of
            when '000'
                SP_main = R[n];
            when '001'
                SP_process = R[n];
    when '00010'
        case SYSm<2:0> of
            when '000'
                PRIMASK = R[n]<0>;
            when `100`
                If CurrentMode == Mode_Thread then CONTROL<1> = R[n]<1>;
```

Exceptions

None.

Notes

Privilege	ARMv6-M always executes as privileged.
IPSR	The currently defined IPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).
EPSR	The currently defined EPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).
Bit positions	The PSR bits are positioned in each PSR according to their position in the larger xPSR composite. This is defined in *The special-purpose program status registers (xPSR)* on page B1-8.

Part C
Debug Architecture

Chapter C1
ARMv6-M Debug

This chapter covers all aspects of debug with respect to ARMv6-M. It is made up of the following sections:

The supported features are a subset of those available in the ARMv7-M profile.

C1.1 Introduction to debug

Debug support is a key element of the ARM architecture. ARMv6-M debug is optional and only available where the Debug Extension is implemented. The Debug Extension is limited to invasive debug (an ability to configure and halt the core using breakpoints, watchpoints or vector catching), and an optional non-invasive PC sampling feature accessible through the Debug Access Port.

Debug is accessed via the Debug Access Port (DAP, see *The Debug Access Port (DAP)* on page C1-4). The DAP allows access to debug resources when the processor is running, halted, or held in reset. When a core is halted, the core is in Debug state.

Note

ARMv6-M restricts core access to the debug resources - see *PPB debug related regions*

As well as the Debug Control Block (DCB) within the System Control Space (SCS), other debug related resources are allocated fixed 4KB address regions within the Private Peripheral Bus (PPB) region of the ARMv6-M system address map:

- Debug Watchpoint and Trace (DWT) provides watchpoint support. Trace is not supported.
- A Breakpoint (BPU) block. This block is a subset of the Flash Patch and Breakpoint (FPB) block available in ARMv7-M. Only breakpoint functionality is supported.
- ROM table. A table of entries providing a mechanism for a debugger to identify the debug infrastructure supported by the implementation.

The address ranges for the DWT, BPU, SCB and DCB are listed in Table C1-1.

Table C1-1 PPB debug related regions

Private Peripheral Bus (address range 0xE0000000 to 0xE00FFFFF)		
Group	**Address Offset Range(s)**[a]	**Notes**
Instrumentation Trace Macrocell (ITM)	`0xE0000000-0xE0000FFF`	Reserved in ARMv6-M
Data Watchpoint and Trace (DWT)	`0xE0001000-0xE0001FFF`	watchpoint support
Breakpoint unit (BPU)	`0xE0002000-0xE0002FFF`	breakpoint support
SCS: System Control Block (SCB)[b]	`0xE000ED00-0xE000ED8F`	Most resources in the SCB are core accessible. AIRCR.VECTCLRACTIVE[c], DFSR and SHCSR are defined as debug resources
SCS: Debug Control Block (DCB)	`0xE000EDF0-0xE000EEFF`	debug control and configuration

Table C1-1 PPB debug related regions (continued)

Private Peripheral Bus (address range 0xE0000000 to 0xE00FFFFF)		
Trace Port Interface Unit (TPIU)	`0xE0040000-0xE0040FFF`	Reserved in ARMv6-M
Embedded Trace Macrocell (ETM)	`0xE0041000-0xE0041FFF`	Reserved in ARMv6-M
ARMv6-M ROM table	`0xE00FF000-0xE00FFFFF`	

a. Debug resources are accessible via the DAP interface only. Access from the core is UNPREDICTABLE with the added restriction that the resources must not be modified (write ignore behavior) on a write access.
b. The SCB is described in *The System Control Block (SCB)* on page B3-7.
c. For a definition of the VECTCLRACTIVE bitfield, see Table B3-7 on page B3-9.

C1.2 The Debug Access Port (DAP)

Debug access is through the Debug Access Port (DAP), an implementation of the *ARM Debug Interface v5 Architecture Specification*. The DAP specification includes details on how a system can be interrogated to determine what debug resources are available, and how to access any ARMv6-M device(s). The ROM table of information as described in Table C1-3 is DAP accessible only and forms part of the Debug Extension. The general format of a ROM table entry is described in Table C1-2.

A debugger can use a DAP interface to interrogate a system for memory access ports (MEM-APs). The BASE register in a memory access port provides the address of the ROM table (or a series of ROM tables within a ROM table hierarchy). The memory access port can then be used to fetch the ROM table entries. See *ARM Debug Interface v5 Architecture Specification* for more information.

Table C1-2 ROM table entry format

Bits	Name	Description
[31:12]	Address offset	Signed base address offset of the component relative to the ROM base address.
[11:2]	Reserved	UNK/SBZP
[1]	Format	1: 32-bit format 0: 8-bit format (not used by ARMv6-M)
[0]	Entry present	1: valid table entry 0: (and bits [31:1] not equal to zero), ignore the table entry[a]

a. `0x00000002` is the recommended null entry for ARMv6-M where a null entry is required before an end of table marker.

For ARMv6-M all address offsets are negative. The entry `0x00000000` indicates the end of table marker.

Table C1-3 ARMv6-M DAP accessible ROM table

Offset	Value	Name	Description
`0x000`	`0xFFF0F003`	SCS	Points to the SCS at 0xE000E000.
`0x004`	`0xFFF02002` or `0xFFF02003`	DWT	Points to the Data Watchpoint and Trace block at `0xE0001000`. Bit [0] is set if a DWT is fitted.
`0x008`	`0xFFF03002` or `0xFFF03003`	BPU	Points to the Breakpoint unit at `0xE0002000`. Bit [0] is set if a BPU is fitted.
`0x00C`	`0x00000000`	end	End-of-table marker. It is IMPLEMENTATION DEFINED whether the table is extended with pointers to other system debug resources. The table entries should terminate with `0x00000000`.
`0x010` to `0xFFC`	if unused, RAZ		For CoreSight compliance requirements, see Appendix A *CoreSight Infrastructure IDs*.

The basic sequence of events to access and enable ARMv6-M debug using a DAP is as follows:

- Enable the power-up bits for the debug logic in the DAP Debug Port control register.
- Ensure the appropriate DAP Memory Access Port control register is enabled for word accesses (this should be the default in a uniprocessor system).
- If halting debug is required, set the C_DEBUGEN bit in the Debug Halting Control and Status Register (DHCSR) – see *Debug Halting Control and Status Register (DHCSR)* on page C1-13.
- If the target is to be halted immediately, set the C_HALT bit in the DHCSR. Read back the S_HALT bit in the DHCSR to ensure the target is halted in Debug state.
- If using watchpoints, set the DWTENA bit in the Debug Exception and Monitor Control Register (DEMCR) – see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-16.

See the *ARM Debug Interface v5 Architecture Specification* for more information on the DAP.

Warning

System control and configuration fields (in particular registers in the SCB) can be changed via the DAP while software is executing. For example, resources designed for dynamic updates can be modified. This can have undesirable side-effects if both the application and debugger are updating the same or related resources. The consequences of updating a running system via a DAP in this manner have no guarantees, and can be worse than UNPREDICTABLE with respect to system behavior.

C1.2.1 General rules applying to debug register access

The Private Peripheral Bus (PPB), address range `0xE0000000` to `0xE0100000`, supports the following general rules:

- The region is defined as Strongly Ordered memory – see *Strongly Ordered memory attribute* on page A3-13 and *Memory access restrictions* on page A3-13.
- Registers are always accessed little endian regardless of the endian state of the processor.
- Debug registers can only be accessed as a word access. Byte and halfword accesses are UNPREDICTABLE.
- The term set means assigning the value to 1 and the term clear(ed) means assigning the value to 0. Where the term applies to multiple bits, all bits assume the assigned value.
- The term disable means assigning the bit value to 0 and the term enable means assigning the bit value to 1.
- Undocumented address space in the PPB is reserved.

C1.3 Overview of the ARMv6-M debug features

The ARMv6-M debug model has control and configuration integrated into the memory map. The Debug Access Port defined in the *ARM Debug Interface v5 Architecture Specification* provides the interface to a host debugger. Debug resources within ARMv6-M are as listed in Table C1-1 on page C1-2.

ARMv6-M supports the following debug related features:

- Reset provision is IMPLEMENTATION DEFINED.
- Core halt. Control register support to halt the processor. This can occur asynchronously by assertion of an external signal, execution of a `BKPT` instruction, or from a debug event (for example when configured to occur on reset, or on entry to a HardFault).
- Step, with or without interrupt masking.
- Run, with or without interrupt masking.
- Register access. The Debug Control Block (DCB) supports debug requests, including reading and writing core registers when halted.
- Access to exception-related information through the System Control Space resources.
- Software breakpoints. The `BKPT` instruction is supported.
- Breakpoint unit support for hardware breakpoints.
- Watchpoint support through the DWT.
- Access to all memory through the DAP.

C1.3.1 Debug authentication

ARM debug notionally supports two generic signals for debug enable and to control invasive versus non-invasive debug as described in Table C1-4.

Table C1-4 ARM debug authentication signals

DBGEN	NIDEN	Invasive debug permitted	Non-invasive debug permitted
LOW	LOW	No	No
LOW	HIGH	No	Yes
HIGH	X	Yes	Yes

For the microcontroller profiles (ARMv6-M or ARMv7-M), the provision of **DBGEN** and **NIDEN** as actual signals is IMPLEMENTATION DEFINED. It is acceptable for **DBGEN** to be considered permanently enabled (**DBGEN** = HIGH), with control deferred to other enable bits within the profile specific debug architecture.

C1.3.2 External debug request

The **EDBGRQ** input is asserted by an external agent to signal an external debug request. An external debug request can cause a debug event and entry to Debug state as described in *Debug event behavior* on page C1-10. The debug event is reported in the DFSR.EXTERNAL status bit, see *Debug Fault Status Register (DFSR)* on page C1-12.

When the processor is in Debug state, the **HALTED** output signal is asserted. **HALTED** reflects the DHCSR.S_HALT bit, see *Debug Halting Control and Status Register (DHCSR)* on page C1-13. The signal can be used as a debug acknowledge for **EDBGRQ**.

EDBGRQ and **HALTED** assert HIGH. **EDBGRQ** is ignored when the core is in Debug state.

C1.3.3 External restart request

It is IMPLEMENTATION DEFINED whether multiprocessing support is provided in the ARMv6-M Debug Extension. An implementation with multi-processing debug support is required to provide the ability to perform a linked restart of multiple cores. Two signals are required to support the multiprocessing restart mechanism:

- a **DBGRESTART** input
- a **DBGRESTARTED** output.

DBGRESTART and DBGRESTARTED

DBGRESTART and **DBGRESTARTED** form a four-phase handshake, as shown in Figure C1-1.

Asserting **DBGRESTART** HIGH causes the core to exit from Debug state. Once **DBGRESTART** is asserted, it must be held HIGH until **DBGRESTARTED** is deasserted. **DBGRESTART** is ignored unless **HALTED** and **DBGRESTARTED** are asserted.

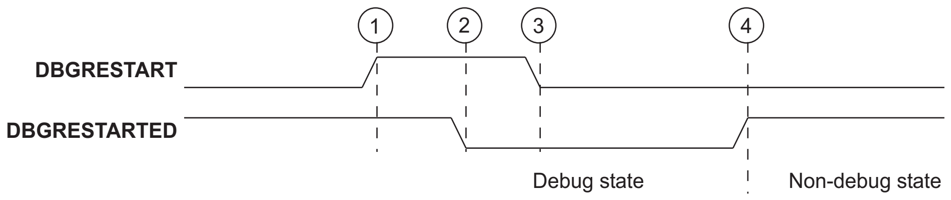

Figure C1-1 DBGRESTART / DBGRESTARTED handshake

Figure C1-1 is diagrammatic only, and no timings are implied. The numbers in Figure C1-1 have the following meanings:

1. If **DBGRESTARTED** is asserted HIGH the peripheral asserts **DBGRESTART** HIGH and waits for **DBGRESTARTED** to go LOW.
2. The processor drives **DBGRESTARTED** LOW to deassert the signal and waits for **DBGRESTART** to go LOW.
3. The peripheral drives **DBGRESTART** LOW to deassert the signal. This is the event that indicates to the processor that it can start the Debug $\rightarrow$ Non-debug state transition phase.

4. The processor leaves Debug state and asserts **DBGRESTARTED** HIGH.

In the process of leaving Debug state the processor clears the **HALTED** signal to LOW. It is IMPLEMENTATION DEFINED when this change occurs relative to the $1 \rightarrow 0$ change in **DBGRESTART** and the $0 \rightarrow 1$ change in **DBGRESTARTED**.

C1.4 Debug and reset

ARMv6-M defines two levels of reset as stated in *Overview of the exceptions supported* on page B1-13:

- a Power-ON Reset (POR)
- a Local Reset.

Note

ARMv6-M does not provide a means to:

- debug a Power-On Reset
- differentiate Power-On Reset from a Local Reset.

Software can initiate a system reset as described in *Reset management* on page B1-32. The reset vector catch control bit (VC_CORERESET) can be used to generate a debug event when the core comes out of reset. A debug event causes the core to enter Debug state when halting debug is enabled.

The following bit fields are reset by a POR but not by a Local Reset:

- fault flags in the DFSR, see *Debug Fault Status Register (DFSR)* on page C1-12
- debug control in the DHCSR, see the notes associated with *Debug Halting Control and Status Register (DHCSR)* on page C1-13
- the vector catch (VC*) configuration bits, see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-16.

For reset and the DWT, see *Register support for the DWT* on page C1-22. For reset and the BPU, see *Register support for the BPU* on page C1-26.

The relationship with the debug logic reset and power control signals described in the DAP recommended external interface (see *ARM Debug Interface v5 Architecture Specification*) is IMPLEMENTATION DEFINED.

C1.5 Debug event behavior

An event triggered for debug reasons is known as a debug event. A debug event will cause one of the following to occur:

- Entry to Debug state. If halting debug is enabled (C_DEBUGEN in the DHCSR, Table C1-9 on page C1-13, is set), captured events will halt the processor in Debug state. See Table B1-7 on page B1-29 for a comprehensive application level fault table.
- If halting debug is disabled, a breakpoint[1] will escalate to a HardFault and other debug events (watchpoints and external debug requests) are ignored.

The Debug Fault Status Register (Table C1-8 on page C1-12) contains status bits for each captured debug event. The bits are write-one-to-clear. These bits are set when a debug event causes the processor to halt or generate an exception. It is IMPLEMENTATION DEFINED whether the bits are updated when an event is ignored.

A summary of halting debug support is provided in Table C1-5.

Table C1-5 Debug related event status

Fault Cause	DFSR Bit Name	Notes
Internal halt request	HALTED	Step command, or a core halt software request
Breakpoint	BKPT	Breakpoint from BKPT instruction or match in BPU
Watchpoint	DWTTRAP	Watchpoint match in DWT
Vector catch	VCATCH	VC_HARDERR or VC_CORERESET event match
External	EXTERNAL	EDBGRQ line asserted

For a description of the vector catch feature, see *Vector catch support* on page C1-17.

If halting debug is disabled and a breakpoint[1] occurs in an NMI or HardFault exception handler, the system locks up with an unrecoverable error. Handling of unrecoverable exceptions in general is described in *Unrecoverable exception cases* on page B1-30.

C1.5.1 Debug stepping

ARMv6-M supports debug stepping. Stepping from Debug state is supported by writing to the C_STEP and C_HALT control bits in the Debug Halt Control and Status Register (see *Debug Halting Control and Status Register (DHCSR)* on page C1-13 for the control bit definitions).

1. This can be due to a BKPT instruction or generated by the BPU, see *Breakpoint unit (BPU) support* on page C1-26.

When C_STEP is set, and C_HALT is cleared in the same or a subsequent register write, the system exits Debug state, the next instruction is executed (stepped) and then the system returns to Debug state. The debugger can optionally set the C_MASKINTS bit in the DHCSR to inhibit (mask) PendSV, Systick and external configurable interrupts from occurring. Where C_MASKINTS is set, permitted exception handlers that activate will execute along with the stepped instruction. See Table C1-6 for a summary of stepping control.

Table C1-6 Debug stepping control using the DHCSR

DHCSR writes[a]			
C_HALT	**C_STEP**	**C_MASKINTS**	**Action**
0	0	0	Exit Debug state and start instruction execution Exceptions activate according to the exception configuration rules.
0	0	1	Exit Debug state and start instruction execution. PendSV, SysTick and external configurable interrupts are disabled, otherwise exceptions activate according to standard configuration rules.
0	1	0	Exit Debug state, step an instruction and halt. Exceptions activate according to the exception configuration rules.
0	1	1	Exit Debug state, step an instruction and halt. PendSV, SysTick and external configurable interrupts are disabled, otherwise exceptions activate according to standard configuration rules.
1	x	x	Remain in Debug state.

a. assumes C_DEBUGEN == 1 and S_HALT == 1 when the write occurs (the system is halted).

Modifying C_STEP or C_MASKINTS while the system is running with halting debug support enabled (C_DEBUGEN == 1, S_HALT == 0) is UNPREDICTABLE.

The values of C_HALT, C_STEP and C_MASKINTS are ignored by hardware and UNKNOWN by software when C_DEBUGEN == 0.

Note

If C_HALT is cleared in Debug state, a subsequent read of S_HALT == '1' means Debug state has been re-entered due to detection of a new debug event.

C1.6 Debug register support in the SCS

The debug provision in the System Control Block consists of two handler-related flag bits (ISRPREEMPT and ISRPENDING) in the Interrupt Control and State Register (*Interrupt Control State Register (ICSR)* on page B3-8), and the Debug Fault Status Register (DFSR).

Additional debug registers are architected in the Debug Control Block as summarized in Table C1-7.

Table C1-7 Debug register region of the SCS

Address	R/W	Name	Function
`0xE000EDF0`	R/W	DHCSR	Debug Halting Control and Status Register
`0xE000EDF4`	WO	DCRSR	Debug Core Register Selector Register
`0xE000EDF8`	R/W	DCRDR	Debug Core Register Data Register
`0xE000EDFC`	R/W	DEMCR	Debug Exception and Monitor Control Register
… to `0xE000EEFF`	…		Reserved for debug extensions

Debug resources are only accessible from the DAP, see Table C1-1 on page C1-2 for more information.

C1.6.1 Debug Fault Status Register (DFSR)

The Debug Fault Status Register as defined in Table C1-8 provides the top level reason why a debug event has occurred. The bits are read/write-one-to-clear. These bits are set when a debug event causes the processor to halt. It is IMPLEMENTATION DEFINED whether the bits are updated when an event is ignored.

Table C1-8 Debug Fault Status Register (0xE000ED30)

Bits[a]	R/W[b]	Name	Function
[31:5]			Reserved
[4]	R/W1C	EXTERNAL	EDBGRQ asserted from outside SCS.
[3]	R/W1C	VCATCH	Vector catch triggered.
[2]	R/W1C	DWTTRAP	Data watchpoint trap.
[1]	R/W1C	BKPT	`BKPT` instruction executed or breakpoint match in the BPU.
[0]	R/W1C	HALTED	Halt generated from a C_HALT or C_STEP request.

a. bits [4:0] are cleared on a power-up reset. They are not cleared by a software initiated (local) reset.
b. R/W1C: Read/Write-one-to-clear.

C1.6.2 Debug Halting Control and Status Register (DHCSR)

The Debug Halting Control and Status Register (DHCSR) controls halting debug.

Table C1-9 Debug Halting Control and Status Register – (0xE000EDF0)

Bits	R/W	Name	Function
[31:16]	W	DBGKEY	Debug Key. The value `0xA05F` must be written to enable write accesses to bits [15:0], otherwise the write access will be ignored. Read behavior of bits [31:16] is as listed below.
[31:26]	-	-	Reserved
[25]	R	S_RESET_ST	Core reset since the last time this bit was read. This is a sticky bit, which clears on read. Multiple reads of the register indicating the bit is set means the core is currently in the reset state.
[24]	R	S_RETIRE_ST	~~Instruction has completed (retired) since last read. This is a sticky bit, which clears on read. This bit can be used to determine if the core is stalled on a load/store or fetch.~~
[23:20]	-	-	Reserved
[19]	R	S_LOCKUP	The core is locked up due to an unrecoverable exception. See *Unrecoverable exception cases* on page B1-30 for details. This bit can only be read as set by a debugger via the DAP port while the core is running and locked up. The bit is cleared on entry to Debug state when the core halts.
[18]	R	S_SLEEP	Core is sleeping. Must set the C_HALT bit to gain control, or wait for an interrupt (`WFI` instruction response) to wake-up the system.
[17]	R	S_HALT	The core is in Debug state.
[16]	R	S_REGRDY	A handshake flag. The bit is cleared to '0' on a write to the Debug Core Register Selector Register and is set to '1' when the transfer to/from the Debug Core Register Data Register is complete. Otherwise the bit is UNKNOWN.
[15:4]	-	-	Reserved
[3]	R/W	C_MASKINTS	Mask PendSV, SysTick and external configurable interrupts when debug is enabled. The bit does not affect NMI. When C_DEBUGEN == '0', this bit is UNKNOWN.

Table C1-9 Debug Halting Control and Status Register – (0xE000EDF0) (continued)

Bits	R/W	Name	Function
[2]	R/W	C_STEP	Step the core in Debug state. When C_DEBUGEN == ’0’, this bit is UNKNOWN.
[1]	R/W	C_HALT	Halt the core. In addition to supporting debug software access, the bit is set to 1 in hardware when triggered by any of the events reported in DFSR[4:1]. The bit reads-as-one in Debug state and is UNKNOWN when C_DEBUGEN == ’0’.
[0]	R/W	C_DEBUGEN	Enable halting debug. C_MASKINTS must be written with 0 when C_DEBUGEN is asserted (written with 1 when previously 0).

Debug Core Register Selector RegisterNotes on Table C1-9 on page C1-13:

- S_RESET_ST is set on every reset (power-on and local resets)
- S_RETIRE_ST and S_HALT clear on reset. If C_HALT and C_DEBUGEN are asserted on reset, S_HALT will be set, and the core will enter Debug state immediately after the reset sequence.
- C_MASKINTS, C_STEP, and C_DEBUGEN are cleared on a power-on reset only.
- Modifying C_STEP or C_MASKINTS while the system is running with halting debug support enabled (C_DEBUGEN == ’1’, S_HALT == ’0’) is UNPREDICTABLE.
- For more information on the use of C_HALT, C_STEP and C_MASKINTS, see *Debug stepping* on page C1-10.

C1.6.3 Debug Core Register Selector Register (DCRSR)

The DCRSR write-only register generates a handshake to the core to transfer the selected register to/from the DCRDR. The DHCSR.S_REGRDY bit is cleared when the DCRSR is written, and remains clear until the core transaction completes. This register is only accessible while in Debug state.

Table C1-10 Debug Core Register Selector Register – (0xE000EDF4)

Bits	R/W	Name	Function
[16]	WO	REGWnR	Write = 1, Read = 0
[4:0]	WO	REGSEL	00000: R0 00001: R1 … 01100: R12 01101: the current SP 01110: LR 01111: DebugReturnAddress() 10000: xPSR / Flags, Execution Number, and state information 10001: MSP (Main SP) 10010: PSP (Process SP) 10100: CONTROL (DCRDR[25:24]), PRIMASK (DCRDR[0]) All unused values reserved

Notes on Table C1-10:

xPSR etc For information on the xPSR see *The special-purpose program status registers (xPSR)* on page B1-8.

Note

A debugger must preserve the Execution Number (IPSR bits) in Debug state, otherwise the behavior is UNPREDICTABLE.

DebugReturnAddress()

The address of the next instruction to be executed on exit from Debug state.

The address reflects the point in the execution stream where the debug event was invoked. For a hardware or a software breakpoint, the address is the breakpointed instruction address.

For all other debug events, including PC watchpoints, the address is that of an instruction which in a sequential execution model is one which executes after the instruction which caused the event, but which itself has not executed. All instructions prior to this instruction in the model have executed.

Note

Bit [0] of DebugReturnAddress() is RAZ/WI. Writing bit [0] does not affect the EPSR.T-bit, which is accessed independently through the xPSR register selection.

C1.6.4 Debug Core Register Data Register (DCRDR)

The DCRDR is used with the DCRSR to provide access to the general-purpose and special-purpose registers in the core. The DCRDR is read or written (depending on DCRSR.REGWnR) to/from the selected register (defined by DCRSR.REGSEL) on a write to the DCRSR.

Note

The xPSR registers are fully accessible using this access method, unlike `using MSR/MRS` instruction access. Illegal values can cause faults to occur.

Table C1-11 Debug Core Register Data Register – (0xE000EDF8)

Bits	R/W	Name	Function
[31:0]	R/W	DBGTMP	Data temporary cache, for reading and writing registers. This register is UNKNOWN on reset or while DHCSR.S_REGRDY == ‘0’ during execution of a DCRSR based transaction that updates the register.

C1.6.5 Debug Exception and Monitor Control Register (DEMCR)

The DEMCR is used to manage exception behavior under debug. It is also used to enable the DWT block, see *Data Watchpoint and Trace (DWT) support* on page C1-18.

Table C1-12 provides the DEMCR details. For a complete list of faults associated with the HardFault exception, see *Fault behavior* on page B1-29.

Table C1-12 Debug Exception and Monitor Control Register – (0xE000EDFC)

Bits	R/W	Name	Function
[31:25]	-	-	Reserved
[24]	R/W	DWTENA[a]	Global enable for all features configured and controlled by the DWT. DWT registers return UNKNOWN values on reads when DWTENA == '0'. In addition, it is IMPLEMENTATION DEFINED whether: a) writes to the DWT unit are ignored while DWTENA is 0 b) DWTENA is cleared by a software initiated reset
[23:11]	-	-	Reserved
[10]	R/W	VC_HARDERR[b]	Debug trap on a HardFault exception. Ignored when C_DEBUGEN is clear.
[9:1]			Reserved
[0]	R/W	VC_CORERESET[b]	Reset Vector Catch. Halt a running system when a Local Reset occurs. Ignored when C_DEBUGEN is clear.

a. Bit known as TRCENA in ARMv7-M. DWTENA can be implemented as RAZ/WI with DWT permanently disabled.

b. The vector catch (VC prefixed) bits are only cleared on a power-up reset.

Vector catch support

Vector catch support is the mechanism used to generate a debug event and enter Debug state when a particular exception occurs.

If C_DEBUGEN in the DHCSR is set, at least one VC* enable bit is set in the DEMCR, and the associated exception activates, then a debug event occurs. This normally causes Debug state to be entered (execution halted) on the first instruction of the exception handler.

Note

Only Debug Fault Status Register bits are available to the debugger to help determine the source of the event (see *Debug Fault Status Register (DFSR)* on page C1-12).

A vector catch guarantees to enter Debug state without executing any additional instructions. However, saved context might include information on a lockup situation or a higher priority pending exception, for example a pending NMI exception detected on reset.

C1.7 Data Watchpoint and Trace (DWT) support

The Data Watchpoint and Trace (DWT) component supports the following features:

- external PC sampling using a PC sample register.
- comparator support for data (watchpoint) and instruction (PC watchpoint) address matching.

Note

The reporting behavior of a PC watchpoint is different from a BKPT instruction or breakpoint detection in a BPU, see Table C1-5 on page C1-10 for details.

C1.7.1 Theory of operation

The PC sampling feature (DWT_PCSR, see *Program counter sampling support*) and watchpoint support operate independently of each other. The number of watchpoints that can be supported is defined in the DWT Control Register (DWT_CTRL, see *Control Register (DWT_CTRL)* on page C1-23). Watchpoint support uses a set of compare, mask and function registers (DWT_COMPx, DWT_MASKx, and DWT_FUNCTIONx). See *Comparator Register (DWT_COMPx)* on page C1-24, *Mask Register (DWT_MASKx)* on page C1-24, and *Function Register (DWT_FUNCTIONx)* on page C1-25 for details.

Watchpoint events result in a core halt and entry to Debug state.

Exception trace support

Exception tracing is not supported in this release of the ARMv6-M Debug Extension.

Program counter sampling support

The DWT Program Counter Sampling Register (DWT_PCSR) is an IMPLEMENTATION DEFINED option in ARMv6-M. The register is defined such that it can be accessed by a debugger without changing the behavior of any code currently executing on the device. This provides a mechanism for coarse-grained non-intrusive profiling of code executing on the core.

The DWT_PCSR is a word-accessible read-only register. Writes to the register are ignored. Byte or halfword reads are UNPREDICTABLE. When the register is read it returns one of the following:

- the address of an instruction *recently executed* by the core
- 0xFFFFFFFF if implemented and the processor is in Debug state, or in a state and mode where non-invasive debug is not permitted
- RAZ/WI if not implemented.

——— **Note** ———

There is no architectural definition of *recently executed.* The delay between an instruction being executed by the core and its address appearing in the DWT_PCSR is not defined. There is no guaranteed relationship between the program counter for a piece of code designed to read the DWT_PCSR and the value read. The DWT_PCSR is intended only for use by an external agent to provide statistical information for code profiling. Read accesses made to the DWT_PCSR directly by the ARM core can return an UNKNOWN value.

A debug agent should not rely on a return value of 0xFFFFFFFF to indicate that the core is halted. The S_HALT bit in the Debug Halting Control and Status Register should be used for this purpose.

———————————

The value read always references a *committed instruction*, where a committed instruction is defined as an instruction which is both fetched and committed for execution. It is IMPLEMENTATION DEFINED whether instructions that do not pass their condition codes are considered as committed instructions. ARM recommends that these instructions are treated as committed instructions.

Implementations must not sample values that reference instructions that are fetched but not committed for execution. The register is read-only, write accesses are ignored. A read access from the DWT_PCSR returns an UNKNOWN value when the DWTENA bit in the Debug Exception and Monitor Control Register is clear.

Comparator support

The DWT_COMPx, DWT_MASKx, and DWT_FUNCTIONx register sets provide the programming interface for the type of match to perform, and the action to take on a match. The number of register sets supported is IMPLEMENTATION DEFINED and can be determined by reading the NUMCOMP field in the DWT_CTRL register.

ARMv6-M supports:

- data address matching and creation of a watchpoint event
- instruction address matching and creation of a PC watchpoint event.

This is controlled by the DWT_FUNCTIONx registers as illustrated in Table C1-13 on page C1-20.

DWT_COMPx contains the reference value (COMP) for the comparator. This value is compared against an input value to determine a match.

For instruction and data address matching the input value is masked. The number of input value bits masked (MASK) is defined in DWT_MASKx[4:0]. The maximum address mask range supported (up to 2GB) is IMPLEMENTATION DEFINED. An address match for a 32-bit instruction must match on the first halfword of the instruction. An address match on only the second halfword of a 32-bit instruction results in UNPREDICTABLE behavior.

——— **Note** ———

The recommended mechanism for generating a breakpoint on a single instruction address is to use the BPU, see *Breakpoint unit (BPU) support* on page C1-26, where supported. The DWT based mechanism must be used to generate a PC matching event on a range of addresses.

Table C1-13 General DWT function support

DWT_FUNCTIONx	Comparator		Function Description/Action
Bits [3:0]	Input[a]	Access	Match(Input, COMP) == TRUE
0000			disabled
0001	-	-	Reserved
0010	-	-	Reserved
0011	-	-	Reserved
0100	Iaddr	-	PC watchpoint event[b]
0101	Daddr	RO[c]	Watchpoint event (optional)
0110	Daddr	WO[c]	Watchpoint event (optional)
0111	Daddr	R/W	Watchpoint event[b]
1xxx	-	-	Reserved

a. Daddr: data access address match. Iaddr: instruction address match.
b. Halt when debug enabled (C_DEBUGEN == 1 and DWTENA == 1). The address of the next instruction to execute following a watchpoint event halt (`DebugReturnAddress()`, see Table C1-10 on page C1-15) is IMPLEMENTATION DEFINED.
c. Support of explicit watchpoint read and write functionality is IMPLEMENTATION DEFINED.

Support of explicit watchpoint read and watchpoint write functionality can be determined by writing the appropriate value to DWT_FUNCTIONx.FUNCTION and reading back the result. Read-only or write-only watchpoint requests convert to read/write watchpoints where the requested feature is not supported.

Comparator support - instruction address matching

Instruction address matching is supported on all comparators. DWT_COMPx must be halfword aligned, and the mask register must be applied by software to the comparator reference value before it is written to DWT_COMPx, otherwise the comparator operation is UNPREDICTABLE.

An instruction address match on the `NOP` instruction is UNPREDICTABLE with respect to whether the event occurs or not.

The comparator behavior is defined as follows:

```
// InstructionAddressMatch()
// ========================

boolean InstructionAddressMatch(bits(32) Iaddr)

    boolean match;

    if DWT_FUNCTION[N]<3:0> == '0100' then                // condition for selecting Iaddr
        // UNPREDICTABLE if COMP does not meet alignment and masking conditions
        mask = ZeroExtend(Ones(UInt(DWT_MASK[N]<4:0>)), 32);
        if !IsZero(DWT_COMP[N] & mask) then UNPREDICTABLE;
        match = (Iaddr & NOT(mask) == DWT_COMP[N]);
    else
        match = FALSE;
    return match;
```

Comparator support - data address matching

For data address compares, the matching address range must be naturally aligned to the size of the memory access memory access, otherwise the match is UNPREDICTABLE. Where the restriction is met, any access within the address range will match. ARMv6-M only supports aligned accesses. Unaligned accesses take a HardFault exception as described in *Alignment support* on page A3-3 and Table B1-7 on page B1-29. Data address matching comparator behavior is defined as follows:

```
~~enumeration sizeofaccess (byte, halfword, word);~~
~~boolean validDaddr;~~       ~~// conditions for selecting Daddr from~~
                           ~~// ...configuraton of the relevant DWT_FUNCTIONx register~~

~~if validDaddr then~~
    ~~match = DataAddrMatch(N, Daddr, Dsize);~~
    ~~return match;~~


~~// DataAddressMatch()~~
~~// ==================~~

~~boolean DataAddrMatch(UInt N, bits(32) address, sizeofaccess size)~~

    ~~// UNPREDICTABLE if COMP isn't properly aligned~~
    ~~mask = ZeroExtend(Ones(UInt(DWT_MASK[N]<4:0>)), 32);~~
    ~~if !IsZero(DWT_COMP[N]<31:0> & mask) then UNPREDICTABLE;~~

    ~~case size of~~
        ~~when word~~
            ~~if DWT_COMP[N]<1:0> != '00' then UNPREDICTABLE;~~
        ~~when halfword~~
            ~~if DWT_COMP[N]<0> != '0' then UNPREDICTABLE;~~
   ~~match = (address & !mask) == DWT_COMP[N]<31:0>);~~
   ~~return match;~~
```

Note

The conditions for selecting `Daddr` is defined in Table C1-13 on page C1-20.

C1.7.2 Register support for the DWT

The DWT is programmed using the registers described in Table C1-14.

Table C1-14 DWT register summary

Address	R/W	Name	Notes
`0xE0001000`	RO	DWT_CTRL	Control register
`0xE000101C`	RO	DWT_PCSR	PC sampling register (optional)
`0xE0001020`	R/W	DWT_COMP0	DWT Comparator Register 0
`0xE0001024`	R/W	DWT_MASK0	DWT Mask Register 0
`0xE0001028`	R/W	DWT_FUNCTION0	DWT Function Register 0
`0xE00010x0/4/8`	R/W	...	Number of DWT register sets defined in DWT_CTRL.NUMCOMP
`0xE0001100`	R/W	DWT_COMP14	where DWT_CTRL.NUMCOMP == 15
`0xE0001104`	R/W	DWT_MASK14	where DWT_CTRL.NUMCOMP == 15
`0xE0001108`	R/W	DWT_FUNCTION14	where DWT_CTRL.NUMCOMP == 15
`0xE0001FB4`[a]	RO		See Appendix A *CoreSight Infrastructure IDs*, Table A-2 on page AppxA-3
`0xE0001FD0` … `0xE0001FFC`	RO		ID space: See Appendix A *CoreSight Infrastructure IDs* for more information.

a. CoreSight compliance for the lock status register is required.

Control Register (DWT_CTRL)

Table C1-15 DWT_CTRL (0xE0001000)

Bits	R/W	Name	Reset Value	Function
[31:28]	RO	NUMCOMP	IMP DEF	Number of comparators available.
[27:0]				Reserved

Program Counter Sample Register (DWT_PCSR)

Table C1-16 DWT_PCSR (0xE000101C)

Bits	R/W	Name	Function
[31:0]	RO	EIASAMPLE	Executed Instruction Address sample value

The register value is UNKNOWN on reset. Bit [0] is RAZ and does not reflect T-bit status as is the case with similar functionality in other profiles.

Comparator Register (DWT_COMPx)

Table C1-17 DWT_COMPx (0xE0001020 etc.)

Bits	R/W	Name	Function
[31:0]	R/W	COMP	Reference value for comparison against the comparator input. See *Comparator support* on page C1-19. The value is UNKNOWN on reset.

Mask Register (DWT_MASKx)

Table C1-18 DWT_MASKx (0xE0001024 etc.)

Bits	R/W	Name	Function
[31:5]	-		Reserved
[4:0]	R/W	MASK	The size of the ignore mask (0-31 bits) applied to address range matching. See *Comparator support* on page C1-19 for the usage model. The mask range is IMPLEMENTATION DEFINED. Writing all 1's to this field and reading it back can be used to determine the maximum mask size supported. The value is UNKNOWN on reset.

Function Register (DWT_FUNCTIONx)

This register controls the operation of the comparator DWT_COMPx.

Table C1-19 DWT_FUNCTIONx (0xE0001028 etc.)

Bits	R/W	Name	Function
[31:25]	-		Reserved
[24]	RO	MATCHED	This bit is set when the associated comparator matches. It indicates that the operation defined by FUNCTION has occurred since the bit was last read. The bit is cleared on a read.
[23:4]			Reserved
[3:0]	R/W	FUNCTION	Select action on comparator match according to Table C1-13 on page C1-20 This field is reset to zero.

C1.8 Breakpoint unit (BPU) support

The Breakpoint Unit (BPU) provides support for breakpoint functionality on instruction fetches.

C1.8.1 Theory of operation

There are two types of register:

- a breakpoint control register BP_CTRL
- breakpoint comparator registers.

The number of instruction address comparators is IMPLEMENTATION DEFINED and can be read from the BP_CTRL register (see *Breakpoint Control Register (BP_CTRL)* on page C1-27).

The Breakpoint Control Register provides a global enable bit for the BPU, along with ID fields indicating the numbers of Breakpoint Comparator registers provided.

Each Breakpoint Comparator Register includes its own enable bit which comes into effect when the global enable bit is set. The comparators match word-aligned addresses (mask out address bits [1:0]) within the Code memory region (1st 512KB of the memory map), and only operate on read accesses.

Address matching can be performed on the upper halfword, lower halfword or both halfwords:

- For 16-bit instructions, halfword matches always generate a breakpoint for the associated instruction.
- For 32-bit instructions, a breakpoint must be configured to match the first halfword or both halfwords of the instruction. It is UNPREDICTABLE whether breakpoint matches on only the address of the second halfword of a 32-bit instruction generate a debug event.

Note

It is IMPLEMENTATION DEFINED whether a breakpoint event is generated when debug is disabled (DHCSR.C_DEBUGEN == 0). When no breakpoint event is generated, the breakpointed instruction exhibits its normal architectural behavior. When a breakpoint event is generated, it will be escalated to HardFault, see *Debug event behavior* on page C1-10.

C1.8.2 Register support for the BPU

The BPU register support is listed in Table C1-20:

Table C1-20 Breakpoint unit register summary

Address	Type	Name	Reference
`0xE0002000`	R/W	BP_CTRL	Breakpoint Control Register
`0xE0002004`			Reserved
`0xE0002008`	R/W	BP_COMP0	Breakpoint Comparator Register 0

Table C1-20 Breakpoint unit register summary (continued)

Address	Type	Name	Reference
…		…	Number of comparators defined in BP_CTRL
`0xE0002040`	R/W	BP_COMP14	where NUM_CODE = 15
`0xE0002FB4`[a]	RO		See Appendix A *CoreSight Infrastructure IDs*, Table A-2 on page AppxA-3
`0xE0002FD0`	RO		ID space: See Appendix A *CoreSight Infrastructure IDs* for more information.
…			
`0xE0002FFC`			

a. CoreSight compliance for the lock status register is required.

See below for details on the FPB support registers.

Breakpoint Control Register (BP_CTRL)

Table C1-21 BP_CTRL (0xE0002000)

Bits	R/W	Name	Function
[31:8]	-	-	Reserved
[7:4]	RO	NUM_CODE	Number of breakpoint comparators Comparators: BP_COMP[NUM_CODE-1:0] NUM_CODE == 0: no comparator support
[1]	R/W	KEY	RAZ on reads, SBO (should-be-one) for writes. If zero written, the write to the register can be ignored.
[0]	R/W	ENABLE	Enable the breakpoint unit when set. This bit is cleared on a power-up reset.

Breakpoint Comparator Registers – (BP_COMPx)

A breakpoint is generated on a comparator match. Breakpoints can be resolved on word or halfword boundaries.

Table C1-22 BP_COMPx breakpoint comparison (0xE0002008-2040)

Bits	R/W	Name	Function
[31:30]	R/W	BP_MATCH	BP_MATCH defines the behavior when the COMP address is matched: 00: no breakpoint matching 01: breakpoint on lower half-word, upper is unaffected 10: breakpoint on upper half-word, lower is unaffected 11: breakpoint on both lower and upper half-words. The field is UNKNOWN on reset.
[29]	-	-	Reserved.
[28:2]	R/W	COMP	Address for breakpoint comparison is '000':COMP:'00' The field is UNKNOWN on reset.
[1]	-	-	Reserved.
[0]	R/W	ENABLE	Comparator is enabled when this bit is set. This bit is cleared on a power-up reset. **Note** The master enable in BP_CTRL must also be set to enable a comparator.

Part D

Appendices

Appendix A
CoreSight Infrastructure IDs

ARMv6-M implementations support SCS, BPU, and DWT blocks along with a ROM table as illustrated in Table C1-3 on page C1-4. The ROM table IDs are based on CoreSight, ARM's system debug architecture. The CoreSight architecture programmers' model is defined in the *CoreSight Architecture Specification* with each 4KB register space subdivided into four sections:

- A component ID (offset `0xFF0` to `0xFFF`)
- A peripheral ID (offset `0xFD0` to `0xFEF`)
- CoreSight management registers (offset `0xF00` to `0xFCF`)
- Device specific registers (offset `0x000` to `0xEFF`)

To determine the topology of the ARMv6-M debug infrastructure, ROM table entries indicate whether a block is present. Presence of an entry guarantees support of the ARMv6-M programming requirements for the entry. Additional functionality requires additional support, where CoreSight is the recommended framework.

The CPUID support in the SCS is the primary ID for detecting an ARMv6-M implementation.

The component ID and peripheral ID register general formats are as shown in Table A-1.

Table A-1 Component and Peripheral ID register formats

Address Offset	Value	Symbol	Name	Reference[a]
`0xFFC`	`0x000000B1`	CID3	Component ID3	Preamble
`0xFF8`	`0x00000005`	CID2	Component ID2	Preamble
`0xFF4`	`0x000000X0`	CID1	Component ID1	bits [7:4] Component Class bits [3:0] Preamble
`0xFF0`	`0x0000000D`	CID0	Component ID0	Preamble
`0xFEC`	`0x000000YY`	PID3	Peripheral ID3	bits [7:4] RevAnd - minor revision field bits [3:0] non-zero == Customer modified block
`0xFE8`	`0x000000YX`	PID2	Peripheral ID2	bits [7:4] Revision bit [3] == 1: JEDEC assigned ID fields bits [2:0] JEP106 ID code [6:4]
`0xFE4`	`0x000000XY`	PID1	Peripheral ID1	bits [7:4] JEP106 ID code [3:0] bits [3:0] Part Number [11:8]
`0xFE0`	`0x000000YY`	PID0	Peripheral ID0	Part Number [7:0]
`0xFDC`	`0x00000000`	PID7	Peripheral ID7	Reserved
`0xFD8`	`0x00000000`	PID6	Peripheral ID6	Reserved
`0xFD4`	`0x00000000`	PID5	Peripheral ID5	Reserved
`0xFD0`	`0x000000YX`	PID4	Peripheral ID4	bits [7:4] 4KB count bits [3:0] JEP106 continuation code

1) For ARMv6-M, all CoreSight registers are accessed as words. Any 8-bit or 16-bit registers defined in the *CoreSight Architecture Specification* are accessed as zero-extended words.

2) For the Value column:
X: CoreSight architected values (fixed bit field, component class, or JEDEC assigned)
Y: IMPLEMENTATION DEFINED

3) The JEDEC defined fields refer to the block designer's JEDEC code. The combination of part number, designer and component class fields must be unique.

4) For more details on the bit fields, see the CoreSight programmers' model in the *CoreSight Architecture Specification*.

a.

The CoreSight management registers are summarized in Table A-2.

Note

The lock mechanism only applies to software access from the core to the affected block. For ARMv6-M, only DAP access is allowed, meaning the lock status and access registers must RAZ.

Table A-2 ARMv6-M and CoreSight management registers

Component Class[a]	Address Offset	Type	Register Name	Notes
0x1	`0xFCC`		MemType	Bits [31:1] RAZ Bit [0][b] is set to 1 to indicate the system memory is accessible via the DAP. The bit is clear when only debug resources are accessible via the DAP.
0x9	`0xFB4`	RO	Lock Status (LSR)	For ARMv6-M, RAZ
0x9	`0xFB0`	WO	Lock Access (LAR)	For ARMv6-M, this register is UNK.
	Recommendation: All reserved space is CoreSight compliant or RAZ See the programmers' model in the *CoreSight Architecture Specification* for a complete CoreSight management register list and register format details.			

a. For information on component classes, see the Component ID register information in *CoreSight Architecture Specification*.

b. For CoreSight compliance, bit[0] == 1 is required in ARMv6-M, along with valid PIDx and CIDx registers as defined in Table A-1 on page AppxA-2. Otherwise, it is IMPLEMENTATION DEFINED whether MemType is implemented as 0x0 or 0x1.

Appendix B
Legacy Instruction Mnemonics

This appendix provides information about the Unified Assembler Language equivalents of older assembler language instruction mnemonics.

It contains the following sections:

B.1 Thumb instruction mnemonics

The following table shows the pre-UAL assembly syntax used for Thumb instructions before the introduction of Thumb-2 technology and the equivalent UAL syntax for each instruction. It can be used to translate correctly-assembling pre-UAL Thumb assembler code into UAL assembler code.

This table is not intended to be used for the reverse translation from UAL assembler code to pre-UAL Thumb assembler code.

In this table, 3-operand forms of the equivalent UAL syntax are used, except in one case where a 2-operand form needs to be used to ensure that the same instruction encoding is selected by a UAL assembler as was selected by a pre-UAL Thumb assembler.

Table B-1 Pre-UAL assembly syntax

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
ADC <Rd>, <Rm>	ADCS <Rd>, <Rd>, <Rm>	
ADD <Rd>, <Rn>, #<imm>	ADDS <Rd>, <Rn>, #<imm>	
ADD <Rd>, #<imm>	ADDS <Rd>, #<imm>	
ADD <Rd>, <Rn>, <Rm>	ADDS <Rd>, <Rn>, <Rm>	
ADD <Rd>, SP	ADD <Rd>, SP, <Rd>	
ADD <Rd>, <Rm>	ADDS <Rd>, <Rd>, <Rm> ADD <Rd>, <Rd>, <Rm>	If <Rd> and <Rm> are both R0-R7, otherwise (and <Rm> is not SP)
ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	ADR form preferred where possible
ADD <Rd>, SP, #<imm>	ADD <Rd>, SP, #<imm>	
ADD SP, #<imm>	ADD SP, SP, #<imm>	
AND <Rd>, <Rm>	ANDS <Rd>, <Rd>, <Rm>	
ASR <Rd>, <Rm>, #<imm>	ASRS <Rd>, <Rm>, #<imm>	
ASR <Rd>, <Rs>	ASRS <Rd>, <Rd>, <Rs>	
B<cond> <label>	B<cond> <label>	
B <label>	B <label>	
BIC <Rd>, <Rm>	BICS <Rd>, <Rd>, <Rm>	
BKPT <imm>	BKPT <imm>	

Table B-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
BL <label>	BL <label>	
BLX <Rm>	BLX <Rm>	<Rm> can be a high register
BX <Rm>	BX <Rm>	<Rm> can be a high register
CMN <Rn>, <Rm>	CMN <Rn>, <Rm>	
CMP <Rn>, #<imm>	CMP <Rn>, #<imm>	
CMP <Rn>, <Rm>	CMP <Rn>, <Rm>	<Rd> and <Rm> can be high registers.
CPS<effect> <iflags>	CPS<effect> <iflags>	
CPY <Rd>, <Rm>	MOV <Rd>, <Rm>	
EOR <Rd>, <Rm>	EORS <Rd>, <Rd>, <Rm>	
LDMIA <Rn>!, <registers>	LDMIA <Rn>, <registers> LDMIA <Rn>!, <registers>	If <Rn> listed in <registers>, otherwise
LDR <Rd>, [<Rn>, #<imm>]	LDR <Rd>, [<Rn>, #<imm>]	<Rn> can be SP
LDR <Rd>, [<Rn>, <Rm>]	LDR <Rd>, [<Rn>, <Rm>]	
LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	<label> form preferred where possible
LDRB <Rd>, [<Rn>, #<imm>]	LDRB <Rd>, [<Rn>, #<imm>]	
LDRB <Rd>, [<Rn>, <Rm>]	LDRB <Rd>, [<Rn>, <Rm>]	
LDRH <Rd>, [<Rn>, #<imm>]	LDRH <Rd>, [<Rn>, #<imm>]	
LDRH <Rd>, [<Rn>, <Rm>]	LDRH <Rd>, [<Rn>, <Rm>]	
LDRSB <Rd>, [<Rn>, <Rm>]	LDRSB <Rd>, [<Rn>, <Rm>]	
LDRSH <Rd>, [<Rn>, <Rm>]	LDRSH <Rd>, [<Rn>, <Rm>]	
LSL <Rd>, <Rm>, #<imm>	MOVS <Rd>, <Rm> LSLS <Rd>, <Rm>, #<imm>	If <imm> == 0, otherwise
LSL <Rd>, <Rs>	LSLS <Rd>, <Rd>, <Rs>	
LSR <Rd>, <Rm>, #<imm>	LSRS <Rd>, <Rm>, #<imm>	
LSR <Rd>, <Rs>	LSRS <Rd>, <Rd>, <Rs>	

Table B-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
MOV <Rd>, #<imm>	MOVS <Rd>, #<imm>	
MOV <Rd>, <Rm>	ADDS <Rd>, <Rm>, #0 MOV <Rd>, <Rm>	If <Rd> and <Rm> are both R0-R7, otherwise
MUL <Rd>, <Rm>	MULS <Rd>, <Rm>, <Rd>	
MVN <Rd>, <Rm>	MVNS <Rd>, <Rm>	
NEG <Rd>, <Rm>	RSBS <Rd>, <Rm>, #0	
ORR <Rd>, <Rm>	ORRS <Rd>, <Rd>, <Rm>	
POP <registers>	POP <registers>	<registers> can include PC
PUSH <registers>	PUSH <registers>	<registers> can include LR
REV <Rd>, <Rn>	REV <Rd>, <Rn	
REV16 <Rd>, <Rn>	REV16 <Rd>, <Rn	
REVSH <Rd>, <Rn>	REVSH <Rd>, <Rn	
ROR <Rd>, <Rs>	RORS <Rd>, <Rd>, <Rs>	
SBC <Rd>, <Rm>	SBCS <Rd>, <Rd>, <Rm>	
STMIA <Rn>!, <registers>	STMIA <Rn>!, <registers>	
STR <Rd>, [<Rn>, #<imm>]	STR <Rd>, [<Rn>, #<imm>]	<Rn> can be SP
STR <Rd>, [<Rn>, <Rm>]	STR <Rd>, [<Rn>, <Rm>]	
STRB <Rd>, [<Rn>, #<imm>]	STRB <Rd>, [<Rn>, #<imm>]	
STRB <Rd>, [<Rn>, <Rm>]	STRB <Rd>, [<Rn>, <Rm>]	
STRH <Rd>, [<Rn>, #<imm>]	STRH <Rd>, [<Rn>, #<imm>]	
STRH <Rd>, [<Rn>, <Rm>]	STRH <Rd>, [<Rn>, <Rm>]	
SUB <Rd>, <Rn>, #<imm>	SUBS <Rd>, <Rn>, #<imm>	
SUB <Rd>, #<imm>	SUBS <Rd>, #<imm>	
SUB <Rd>, <Rn>, <Rm>	SUBS <Rd>, <Rn>, <Rm>	
SUB SP, #<imm>	SUB SP, SP, #<imm>	

Table B-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
SWI <imm>	SVC <imm>	
SXTB <Rd>, <Rm>	SXTB <Rd>, <Rm>	
SXTH <Rd>, <Rm>	SXTH <Rd>, <Rm>	
TST <Rn>, <Rm>	TST <Rn>, <Rm>	
UXTB <Rd>, <Rm>	UXTB <Rd>, <Rm>	
UXTH <Rd>, <Rm>	UXTH <Rd>, <Rm>	

B.2 Pre-UAL pseudo-instruction NOP

In pre-UAL assembler code, `NOP` is a pseudo-instruction, equivalent to `MOV R8,R8` in Thumb code.

Assembling the `NOP` mnemonic as UAL will not change the functionality of the code, but will change:

- the instruction encoding selected
- the architecture variants on which the resulting binary will execute successfully, because the Thumb version of the `NOP` instruction was introduced in ARMv6T2.

To avoid the change in Thumb code, replace `NOP` in the assembler source code with `MOV R8,R8`, before assembling as UAL.

Note

The pre-UAL pseudo-instruction is different for ARM code where it is equivalent to `MOV R0,R0`.

Appendix C
Deprecated Features in ARMv6-M

Some features of the Thumb instruction set are deprecated. Deprecated features affecting instructions supported by ARMv6-M are as follows:

- use of the PC as `<Rd>` or `<Rm>` in a 16-bit `ADD` (SP plus register) instruction
- use of the SP as `<Rm>` in a 16-bit `CMP` (register) instruction
- use of `<Rn>` as the lowest-numbered register in the register list of a 16-bit STM instruction with base register writeback
- use of `MOV` (register) instructions in which both `<Rd>` and `<Rm>` are the SP or PC.

Appendix D
ARMv7-M Differences

ARMv6-M is derived from and software compatible at the binary level[1] with ARMv7-M (ARMv6-M to ARMv7-M only). The simplifications introduced to ARMv6-M affect the application level, system level and debug capabilities. ARMv6-M is also upwards compatible at the application level with ARMv4T, ARMv5T and ARMv6 Thumb code. By understanding the similarities and differences, it is possible to minimize the effort in supporting software across the range of Thumb devices, or to generate a system architecture allowing straightforward migration from ARMv6-M to ARMv7-M.

For a complete description of the ARMv7-M architecture, see the *ARMv7-M Architecture Reference Manual*.

1. Compatibility assumes the system level support, for example memory provision plus the number and assignment of interrupts, is the same or a superset of the ARMv6-M case, and that the initialization assures compatibility of ARMv6-M reserved registers (CCR.STKALIGN = 1 etc.).

D.1 Instruction support

ARMv6-M and ARMv7-M execute instructions in Thumb state only. ARMv6-M supports the 16-bit Thumb instruction set as defined for ARMv6 plus the following:

- the extended range form of the `BL` instruction
- an M-profile version of the 16-bit `CPS` system control instruction
- the M-profile versions of the 32-bit `MRS`, and `MSR` system control instructions
- the 32-bit `DMB`, `DSB` and `ISB` barrier instructions
- the 16-bit forms of the NOP-compatible hint instructions (`NOP`, `SEV`, `WFE`, `WFI` and `YIELD`).

The ARMv7-M Instruction Set Architecture (ISA) consists of all of the above plus a broad set of 32-bit load, store and data processing instructions.

Note

ARMv6-M does not support exclusive access instructions (`LDREX{B,H}`, `STREX{B,H}`) or any form of atomic swap instruction. Software needs to take account of this in multiprocessing environments which use shared memory.

D.2 Programmers' model support

ARMv6-M supports privileged execution only, always allowing configuration and control of the system resources. ARMv7-M supports privileged and unprivileged execution. While both profiles share a common exception entry/exit model, ARMv6-M has limitations in its support of exception types, and how they can be configured and controlled. A general assumption is that faults are considered fatal in ARMv6-M with very limited support for fault recovery. Table D-1 provides a comparison of programmers' model features:

Table D-1 Programmers' model feature comparison

ARMv6-M	ARMv7-M
Privileged execution only	Privileged and unprivileged execution supported
IMPLEMENTATION DEFINED reset model (supports single reset domain)	Key resource reset differentiated between power-on, software-controlled, and debug resource domains.
Vector table fixed at address 0x00000000	Configurable vector table base address (address 0x000000000 on reset)
Reset, NMI and HardFault fixed priority exceptions	Reset, NMI and HardFault fixed priority exceptions
~~OS Extension adds SVCall, PendSV and~~ SysTick exception support	~~SVCall, PendSV and~~ SysTick ~~exceptions~~ part of the base architecture
~~SP_main stack with optional SP_Process second stack as part of the OS Extension~~	~~SP_main and SP_process both part of the base architecture~~
Support for 4 levels of priority (2 bits), no priority grouping	8-bit priority support including priority grouping
Reduced exception priority management: PRIMASK special-purpose register. No support for changing configurable exceptions when they are active.	Dynamic exception priority control: PRIMASK, FAULTMASK, BASEPRI special-purpose registers. Dynamic management of configurable exceptions supported.
All faults reported as HardFault[a]	MemManage, UsageFault and BusFault exceptions with support for escalation to HardFault
No fault status information (FSR/FAR registers) other than through debug support (DFSR). Faults considered fatal.	FAR/FSR support as applicable for all fault exceptions including debug. Supports recoverable faults.
Stack alignment is mandated, see *Stack alignment on exception entry* on page B1-20.	Stack alignment is a configurable option, controlled by the CCR.STKALIGN bit.

Table D-1 Programmers' model feature comparison (continued)

ARMv6-M	ARMv7-M
Late-arrival exception selection on exception entry permitted	Late-arrival exception selection on exception entry permitted
No tail-chaining support for exception return	Tail-chaining allowed on exception return
No validity checks on exception return	Exception return validity checks
Up to 32 external interrupts in the NVIC	Up to 496 external interrupts in the NVIC
Default memory map only	Default memory map with MPU protected memory management option
A simplified set of memory-mapped system control registers in the System Control Space. Bit fields and whole registers in some cases become reserved in line with the reduced functionality. Debug support is in general both reduced and only available via the DAP.	See the ARMv7-M architecture (System Control Space registers as listed in the System memory Map chapter, Part B) for a complete listing of the base architecture support, and the equivalent sections in Part C for debug register support.
Debug Extension supports halting debug only (Debug state, no debug monitor support)	Debug support part of the base architecture (Debug monitor and halting debug)

a. In ARMv6-M all synchronous faults are handled as a HardFault and can be considered as escalated faults with respect to ARMv7-M.

D.2.1 Power management

The events which wake up a `WFE` can be controlled in ARMv7-M using the System Control Register SEVONPEND bit. If this bit is set, the transition of any interrupts, whether enabled or not, from the Inactive to Pending state will cause an event. The register includes two other control bits associated with power management.

The System Control Register is reserved in ARMv6-M.

D.3 Memory model support

ARMv6-M and ARMv7-M share a common default memory map, with the qualification that vendor system space (Vendor_SYS) from address `0xE0000000` to `0xFFFFFFFF` is reserved in ARMv6-M. ARMv7-M also supports the Protected Memory System Architecture (PMSAv7) as an architecture option.

ARMv7-M supports a continuation model for the `LDM` and `STM` instructions using the ICI bits in the EPSR. The ICI bits are not supported in ARMv6-M. If the instruction aborts due to a fault, it will restart rather than continue on return from the exception. For this reason, ARMv6-M does not support use of Device or Strongly Ordered memory with multi-word load or store instructions.

D.3.1 Alignment support

ARMv6-M only supports naturally aligned memory accesses for 16-bit halfword and 32-bit word accesses using the `LDR`, `LDRH`, `LDRSH`, `STR` and `STRH` instructions. ARMv7-M supports unaligned accesses from these instructions.

——— **Note** ———

ARMv7-M has additional T variant instructions, not supported in ARMv6-M, that also provide unaligned accesses.

———————————

D.3.2 Endian support

A configurable endian model (see *Endian support* on page A3-4) is supported by both ARMv6-M and ARMv7-M. ARMv6-M supports endian choice as a build option. ARMv7-M endian support can be a build option or statically configured on reset.

ARMv6-M and ARMv7-M only support instruction fetches in little endian format. Where a big endian instruction format is required, byte swapping within a halfword is required in the bus fabric. The byte swap is required for instruction fetches only and must not occur on data accesses.

For example, for instruction fetches over a 32-bit bus:

```
PrefetchInstr<31:24> -> PrefetchInstr<23:16>
PrefetchInstr<23:16> -> PrefetchInstr<31:24>
PrefetchInstr<15:8> -> PrefetchInstr<7:0>
PrefetchInstr<7:0> -> PrefetchInstr<15:8>
```

D.3.3 Exclusive access support

ARMv6-M does not support exclusive access instructions.

ARMv7-M supports the `CLREX`, `LDREX`, `LDREXB`, `LDREXH`, `STREX`, `STREXB` and `STREXH` instructions.

D.3.4 Cache support

ARMv6-M and ARMv7-M only support memory-mapped system caches.

D.3.5 Protected Memory System Architecture (PMSA) support

ARMv6-M does not support PMSA. All associated Memory Protection Unit (MPU) registers are reserved.

D.4 System Control Space register support

Many registers or register fields present in ARMv7-M are reserved in ARMv6-M. Reserved fields relate to features not supported, or configurable features in ARMv7-M that are fixed in ARMv6-M.

D.4.1 Reserved registers in ARMv6-M

The following registers are defined in ARMv7-M and reserved in ARMv6-M:

- the System Control Register at address 0xE000ED10
- the Configuration and Control Register[1] at address 0xE000ED14
- the System Handler Priority Register 1 at address 0xE000ED18
- the CPU attribute ID region from address 0xE000ED40 to 0xE000ED7F
- the Coprocessor Access Register at address 0xE000ED88
- the Software Trigger Interrupt Register at address 0xE000EF00
- the Configurable Fault Status Registers at address 0xE000ED28
- the HardFault Status Register at address 0xE000ED2C
- the MemManage Address Register at address 0xE000ED34
- the BusFault Address Register at address 0xE000ED38
- the Auxiliary Fault Status Register at address 0xE000ED3C
- the MPU register address space from 0xE000ED90 to 0xE000EDEF.

For ARMv6-M, support of stack alignment (see *Stack alignment on exception entry* on page B1-20), unaligned access trapping (see *Alignment support* on page A3-3) and priority grouping (see *Priority grouping* on page B1-16) are fixed in hardware with no programming interface.

CPU attribute ID registers

ARMv6-M does not support the CPU attribute ID registers.

D.4.2 General Fault Status Registers

ARMv6-M only supports fault status information (the SHCSR and DFSR) as part of the Debug Extension. All other fault status resources in ARMv7-M are reserved in ARMv6-M.

1. The STKALIGN and UNALIN_TRP functions are permanently enabled in ARMv6-M. The corresponding bits in the ARMv7-M CCR register can be considered as implicitly fixed at 1. Similarly, the AIRCR.PRIGROUP field is assumed to RAZ.

HardFault is used for all fault conditions on ARMv6-M. The different fault categories offered by ARMv7-M (MemManage, BusFault and UsageFault) are always escalated to HardFault in ARMv6-M. See *Priority escalation* on page B1-16 for details.

D.4.3 System timer support

ARMv7-M includes an architected system timer – SysTick. ARMv6-M supports a compatible timer ~~in the OS Extension.~~ The ARMv6-M version does not support an external reference clock source.

D.4.4 Nested Vectored Interrupt Controller (NVIC) support

ARMv6-M supports a compatible NVIC to that supported in ARMv7-M. The only differences are:

- ARMv6-M only supports up to 32 external interrupts. The Interrupt Controller Type register is reserved in ARMv6-M.
- ARMv6-M does not support an Active Bit Register, the register location is reserved.

D.5 Debug support

ARMv7-M supports the Debug Monitor exception as well as halting debug and the associated Debug state. ARMv6-M only supports halting debug, and only when the Debug Extension is present.

Debug resources in ARMv7-M are generally accessible from the core or Debug Access Port (DAP). Debug resources in ARMv6-M are only accessible from the core where they support an application level usage model.

ARMv6-M offers a simpler set of debug features and options to ARMv7-M:

- debug is only provided with the Debug Extension
- halting debug support only - no Debug Monitor exception
- software breakpoint support using the `BKPT` instruction
- hardware breakpoint support using a Breakpoint Unit (BPU) - name changed from Flash Patch and Breakpoint Unit (FPB) in ARMv7-M to reflect the reduced functionality
- watchpoint support (data access address matching only) - no data value watchpoint support
- optional PC sampling register (PCSR) - no other cycle counter or trace support in the DWT
- no Instrumentation Trace Macrocell (ITM) support
- no Embedded Trace Macrocell (ETM) or the associated Trace Port Interface Unit (TPIU) support.

D.5.1 Debug and reset in ARMv6-M

It is IMPLEMENTATION DEFINED whether the power-on reset domain is the same or different from the reset domain under software control.

It is IMPLEMENTATION DEFINED whether the debug reset domain is the same or different from the core's power-on reset domain.

Appendix E
Pseudocode definition

This appendix provides a formal definition of the pseudocode used in this book, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

E.1 Instruction encoding diagrams and pseudocode

Instruction descriptions in this book contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or after only a condition check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is UNPREDICTABLE.
- A named single bit or a bit within a named multi-bit field.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction.

The execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagrams match. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. (The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and to be executed as NOPs.)
2. If the operation pseudocode for the matching encoding diagrams starts with a condition check, perform that condition check. If the condition check fails, abandon this execution model and treat the instruction as a NOP. (If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition check.)
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field within its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit(s) from the bit pattern of the instruction.

   In a few cases, the encoding diagram contains more than one bit or field with the same name. When this occurs, the values of all of those bits or fields are expected to be identical, and the encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if this is not the case. This function returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case (most commonly one indicating that it is UNPREDICTABLE). If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as UNPREDICTABLE.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode (if present) performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

E.1.1 Pseudocode

The pseudocode provides precise descriptions of what instructions do. Instruction fields are referred to by the names shown in the encoding diagram for the instruction.

The pseudocode is described in detail in the following sections.

E.2 Limitations of pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses see *Memory access order* on page A3-16.
- The pseudocode statements `UNDEFINED`, `UNPREDICTABLE` and `SEE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:
  - Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
  - No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

  For more information see *Simple statements* on page AppxE-17.

E.3 Data Types

This section describes:

- *General data type rules*
- *Bitstrings*
- *Integers* on page AppxE-6
- *Reals* on page AppxE-6
- *Booleans* on page AppxE-6
- *Enumerations* on page AppxE-6
- *Lists* on page AppxE-7
- *Arrays* on page AppxE-8.

E.3.1 General data type rules

ARM Architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- bitstring
- integer
- boolean
- real
- enumeration
- list
- array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1`, `y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y` and `z` to have types integer, length-1 bitstring and boolean respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

These data types are described in more detail in the following sections.

E.3.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum allowed length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in the bitstring for clarity.

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons. See *Equality and non-equality testing* on page AppxE-11 for details.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length N is bit N-1 and its rightmost bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of registers, memory locations, instructions, and so on. All of the remaining data types are abstract.

E.3.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it should be written as `-0x80000000`.

E.3.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point (so `0` is an integer constant, but `0.0` is a real constant).

E.3.5 Booleans

A boolean is a logical true or false value.

The type name for booleans is `boolean`. This is not the same type as `bit`, which is a length-1 bitstring.

Boolean constants are `TRUE` and `FALSE`.

E.3.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration SRType (SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX);
```

An enumeration always contains at least one symbolic constant, and symbolic constants are not allowed to be shared between enumerations.

Enumerations must be declared explicitly, though a variable of an enumeration type can be declared implicitly as usual by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its type name, and the symbolic constants are its possible constants.

Note

Booleans are basically a pre-declared enumeration:

```
enumeration boolean {FALSE, TRUE};
```

that does not follow the normal naming convention and that has a special role in some pseudocode constructs, such as `if` statements.

E.3.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, such as:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this particular list is the return type of the function `Shift_C()` that performs a standard ARM shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than parentheses. These are:

- Bitstring extraction operators, which use lists of bit numbers or ranges of bit numbers surrounded by angle brackets "<...>".
- Array indexing, which uses lists of array indexes surrounded by square brackets "[...]".
- Array-like function argument passing, which uses lists of function arguments surrounded by square brackets "[...]".

Each combination of data types in a list is a separate type, with type name given by just listing the data types (that is, `(bits(32),bit)` in the above example). The general principle that types can be declared by assignment extends to the types of the individual list items within a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n` and `(shift_t,shift_n)` to be of types `bits(2)`, `integer` and `(bits(2),integer)` respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as "`abc.shift`" and "`abc.amount`". This sort of qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the above definition of `ShiftSpec`, `ShiftSpec` and `(bits(2),integer)` are two different names for the same type, not the names of two different types. In order to avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to may be written as "`-`" to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, like `('00', 0)` in the above example.

E.3.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges (represented by the lower inclusive end of the range, then "`..`", then the upper inclusive end of the range). For example:

```
enumeration PhysReg {
    PhysReg_R0,    PhysReg_R1,     PhysReg_R2,    PhysReg_R3,
    PhysReg_R4,    PhysReg_R5,     PhysReg_R6,    PhysReg_R7,
    PhysReg_R8,    PhysReg_R9,     PhysReg_R10,   PhysReg_R11,
    PhysReg_R12,   PhysReg_SP_Process,            PhysReg_SP_Main,
    PhysReg_LR,    PhysReg_PC};

array bits(32) _R[PhysReg];

array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because enumerations always contain at least one symbolic constant and integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address,size]` or `Element[i,type]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and vector element processing.

E.4 Expressions

This section describes:

- *General expression syntax*
- *Operators and functions - polymorphism and prototypes* on page AppxE-10
- *Precedence rules* on page AppxE-10.

E.4.1 General expression syntax

An expression is one of the following:

- a constant
- a variable, optionally preceded by a data type name to declare its type
- the word `UNKNOWN` preceded by a data type name to declare its type
- the result of applying a language-defined operator to other expressions
- the result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is stated to read as 0 and ignore writes, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like "`bits(32) UNKNOWN`" indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software. (This was called an UNPREDICTABLE value in previous ARM Architecture documentation. It is related to but not the same as `UNPREDICTABLE`, which says that the entire architectural state becomes similarly unspecified.)

A subset of expressions are assignable. That is, they can be placed on the left-hand side of an assignment. This subset consists of:

- Variables
- The results of applying some operators to other expressions. The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. (For example, those circumstances might include one or more of the expressions the operator operates on themselves being assignable expressions.)
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type. This is determined by:

- For a constant, the syntax of the constant.
- For a variable, there are three possible sources for the type
  - its optional preceding data type name

  - a data type it was given earlier in the pseudocode by recursive application of this rule
  - a data type it is being given by assignment (either by direct assignment to it, or by assignment to a list of which it is a member).

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator.
- For a function, the definition of the function.

E.4.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each of the resulting forms of an operator or function has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, and so on, in the prototype definition.

E.4.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables and function invocations are evaluated with higher priority than any operators using their results.
2. Expressions on integers follow the normal *exponentiation before multiply/divide before add/subtract* operator precedence rules, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all allowable precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

E.5 Operators and built-in functions

This section describes:

- *Operations on generic types*
- *Operations on booleans*
- *Bitstring manipulation*
- *Arithmetic* on page AppxE-14.

E.5.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values `x` and `y` of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison with a bitstring constant that includes `'x'` bits as well as `'0'` and `'1'` bits is permitted. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`. This special form is also permitted in the implied equality comparisons in `when` parts of `case ... of ...` structures.

Conditional selection

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

E.5.2 Operations on booleans

If `x` is a `boolean`, then `!x` is its logical inverse.

If `x` and `y` are booleans, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

If `x` and `y` are booleans, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

If `x` and `y` are booleans, then `x ^ y` is the result of exclusive-ORing them together.

E.5.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and top bit

If x is a bitstring, the bitstring length function Len(x) returns its length as an integer, and TopBit(x) is the leftmost bit of x (= x<Len(x)-1> using bitstring extraction.

Bitstring concatenation and replication

If x and y are bitstrings of lengths N and M respectively, then x:y is the bitstring of length N+M constructed by concatenating x and y in left-to-right order.

If x is a bitstring and n is an integer with n > 0, Replicate(x,n) is the bitstring of length n*Len(x) consisting of n copies of x concatenated together and:

- Zeros(n) = Replicate('0',n)
- Ones(n) = Replicate('1',n)

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is x<integer_list>, where x is the integer or bitstring being extracted from, and <integer_list> is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in <integer_list>.

In x<integer_list>, each of the integers in <integer_list> must be:

- >= 0
- < Len(x) if x is a bitstring.

The definition of x<integer_list> depends on whether integer_list contains more than one integer. If it does, x<i,j,k,...,n> is defined to be the concatenation:

```
x<i> : x<j> : x<k> : ... : x<n>
```

If integer_list consists of just one integer i, x<i> is defined to be:

- if x is a bitstring, '0' if bit i of x is a zero and '1' if bit i of x is a one.
- if x is an integer, let y be the unique integer in the range 0 to 2^(i+1)-1 that is congruent to x modulo 2^(i+1). Then **x<i>** is '0' if y < 2^i and '1' if y >= 2^i.

Loosely, this second definition treats an integer as equivalent to a sufficiently long 2's complement representation of it as a bitstring.

In <integer_list>, the notation i:j with i >= j is shorthand for the integers in order from i down to j, both ends inclusive. For example, instr<31:28> is shorthand for instr<31,30,29,28>.

The expression x<integer_list> is assignable provided x is an assignable bitstring and no integer appears more than once in <integer_list>. In particular, x<i> is assignable if x is an assignable bitstring and 0 <= i < Len(x).

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31>.

Logical operations on bitstrings

If `x` is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of `x`.

If `x` and `y` are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of `x` and `y` together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of `x` that are ones.

Testing a bitstring for being all zero or all ones

If `x` is a bitstring, `IsZero(x)` produces `TRUE` if all of the bits of `x` are zeros and `FALSE` if any of them are ones, and `IsZeroBit(x)` produces `'1'` if all of the bits of `x` are zeros and `'0'` if any of them are ones. IsOnes(x) and IsOnesBit(x) work in the corresponding way. So:

```
IsZero(x)    = (BitCount(x) == 0)

IsOnes(x)    = (BitCount(x) == Len(x))

IsZeroBit(x) = if IsZero(x) then '1' else '0'

IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If `x` is a bitstring, and N = Len(x):

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)` is the number of zero bits at the left end of x, in the range 0 to N.
- `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N-1.

Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x,i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x,i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x,i) = Zeros(i-Len(x)) : x
```

If x is a bitstring and i is an integer, then SignExtend(x,i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == Len(x), then SignExtend(x,i) = x, and if i > Len(x), then:

```
SignExtend(x,i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either ZeroExtend(x,i) or SignExtend(x,i) in a context where it is possible that i < Len(x).

Converting bitstrings to integers

If x is a bitstring, SInt(x) is the integer whose 2's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

UInt(x) is the integer whose unsigned representation is x:

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

E.5.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Unary plus, minus and absolute value

If x is an integer or real, then +x is x unchanged, -x is x with its sign reversed, and ABS(x) is the absolute value of x. All three are of the same type as x.

Addition and subtraction

If x and y are integers or reals, x+y and x-y are their sum and difference. Both are of type integer if x and y are both of type integer, and real otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If x and y are bitstrings of the same length N = Len(x) = Len(y), then x+y and x-y are the least significant N bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>

x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If x is a bitstring of length N and y is an integer, x+y and x-y are the bitstrings of length N defined by x+y = x + y<N-1:0> and x-y = x - y<N-1:0>. Similarly, if x is an integer and y is a bitstring of length M, x+y and x-y are the bitstrings of length M defined by x+y = x<M-1:0> + y and x-y = x<M-1:0> - y.

Comparisons

If x and y are integers or reals, then x == y, x != y, x < y, x <= y, x > y, and x >= y are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing boolean results. In the case of == and !=, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If x and y are integers or reals, then x * y is the product of x and y, of type integer if both x and y are of type integer and otherwise of type real.

Division and modulo

If x and y are integers or reals, then x / y is the result of dividing x by y, and is always of type real.

If x and y are integers, then x DIV y and x MOD y are defined by:

```
x DIV y = RoundDown(x / y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any x / y, x MOD y, or x DIV y in any context where y can be zero.

Square Root

If x is an integer or a real, Sqrt(x) is its square root, and is always of type real.

Rounding and aligning

If x is a real:

- RoundDown(x) produces the largest integer n such that n <= x.
- RoundUp(x) produces the smallest integer n such that n >= x.
- RoundTowardsZero(x) produces RoundDown(x) if x > 0.0, 0 if x == 0.0, and RoundUp(x) if x < 0.0.

If x and y are integers, Align(x,y) = y * (x DIV y) is an integer.

If x is a bitstring and y is an integer, Align(x,y) = (Align(UInt(x),y))<Len(x)-1:0> is a bitstring of the same length as x.

It is a pseudocode error to use either form of Align(x,y) in any context where y can be 0. In practice, Align(x,y) is only used with y a constant power of two, and the bitstring form used with y = 2^n has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If n is an integer, 2^n is the result of raising 2 to the power n and is of type real.

If x and n are integers, then:

- x << n = RoundDown(x * 2^n)
- x >> n = RoundDown(x * 2^(-n)).

Maximum and minimum

If x and y are integers or reals, then Max(x,y) and Min(x,y) are their maximum and minimum respectively. Both are of type integer if both x and y are of type integer and of type real otherwise.

E.6 Statements and program structure

This section describes the control statements used in the pseudocode.

E.6.1 Simple statements

The following simple statements must all be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type the function prototype line declared.

UNDEFINED

The statement:

```
UNDEFINED;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

The statement:

```
UNPREDICTABLE;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is not architecturally defined and must not be relied upon by software. It must not constitute a security hole or halt or hang the system, and must not be promoted as providing any useful information to software.

SEE...

The statement:

```
SEE <reference>;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

IMPLEMENTATION_DEFINED

The statement:

```
IMPLEMENTATION_DEFINED <text>;
```

indicates a special case that specifies that the behavior is IMPLEMENTATION DEFINED. Following text can give more information.

SUBARCHITECTURE_DEFINED

The statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

indicates a special case that specifies that the behavior is SUBARCHITECTURE DEFINED. Following text can give more information.

E.6.2 Compound statements

Indentation is normally used to indicate structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if ... then ... else ...

A multi-line `if ... then ... else ...` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

The `else` and its following statements are optional.

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and (if present) the `else` part, as follows:

```
if <boolean_expression> then <statement 1>

if <boolean_expression> then <statement 1> else <statement A>

if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if ... then ... else ...` expression.

repeat ... until ...

A `repeat ... until ...` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    ...
    <statement n>
until <boolean_expression>;
```

while ... do

A `while ... do` structure takes the form:

```
while <boolean_expression> do
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

for ...

A `for ...` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

case ... of ...

A `case ... of ...` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        ...
        <statement n>
    ... more "when" groups ...
    otherwise
        <statement A>
        <statement B>
        ...
        <statement Z>
```

where `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. See *Equality and non-equality testing* on page AppxE-11 for details.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

where the `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function is similar, but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

E.6.3 Comments

Two styles of pseudocode comment exist:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

E.7 Miscellaneous helper procedures and functions

The functions described in this section are not part of the pseudocode specification. They are *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. Each has a brief description and a pseudocode prototype. Some have had a pseudocode definition added.

E.7.1 ALUWritePC()

This procedure writes a value to the PC with the correct semantics for such a write by the `ADD (register)` and `MOV (register)` data-processing instructions.

```
ALUWritePC(bits(32) value)
```

E.7.2 ArchVersion()

This function returns the major version number of the architecture.

```
integer ArchVersion()
```

E.7.3 BadReg()

This function performs the check for the register numbers 13 and 15 that are disallowed for many Thumb register specifiers.

```
boolean BadReg(integer n)
    return n == 13 || n == 15;
```

E.7.4 BigEndian()

This function returns `TRUE` if load/store operations are currently big-endian, and `FALSE` if they are little-endian.

```
boolean BigEndian()
```

E.7.5 BigEndianReverse()

This function is used to reverse the bytes in a value where a big endian access is required.

E.7.6 BranchWritePC()

This procedure writes a value to the PC with the correct semantics for such writes by simple branches - that is, just a change to the PC in all circumstances.

```
BranchWritePC(bits(32) value)
```

E.7.7 BreakPoint()

This procedure causes a debug breakpoint to occur.

E.7.8 BXWritePC()

This procedure writes a value to the PC with the correct semantics for such writes by interworking instructions. That is, with `BX`-like interworking behavior in all circumstances.

```
BXWritePC(bits(32) value)
```

Note

The M profile only supports the Thumb execution state. An attempt to change the instruction execution state causes an exception.

E.7.9 CallSupervisor()

In the M profile, this procedure causes an SVCall exception.

E.7.10 CheckPermissions()

Used with ValidateAddress() to check memory access permission and raise an access violation exception where appropriate.

E.7.11 ClearExclusiveByAddress()

Return a local monitor to its Open Access state where the address tag matches. It is IMPLEMENTATION DEFINED whether an associated global monitor is cleared.

E.7.12 ClearExclusiveMonitors()

This procedure clears the local monitor used by the load/store exclusive instructions. It is IMPLEMENTATION DEFINED whether an associated global monitor is cleared.

E.7.13 ConditionPassed()

This function performs the condition test for an instruction, based on:

- the two Thumb conditional branch encodings (encodings T1 andT3 of the B instruction)
- the current values of the xPSR.IT[7:0] bits for other Thumb instructions.

```
boolean ConditionPassed()
```

E.7.14 DataMemoryBarrier()

This procedure produces a Data Memory Barrier.

```
DataMemoryBarrier(bits(4) option)
```

E.7.15 DataSynchronizationBarrier()

This procedure produces a Data Synchronization Barrier.

```
DataSynchronizationBarrier(bits(4) option)
```

E.7.16 Deactivate()

This function is used with PopStack() for exception return.

E.7.17 DecodeImmShift(), DecodeRegShift()

These functions perform the standard *2-bit type, 5-bit amount* and *2-bit type* decodes for immediate and register shifts respectively. See *Shift operations* on page A6-10.

E.7.18 DefaultAttrs()

Determine memory attributes from the system memory map.

E.7.19 DefaultDecode()

Used with ValidateAddress() to determine the memory attributes from the MPURASR

E.7.20 EncodingSpecificOperations()

This procedure invokes the encoding-specific pseudocode for an instruction encoding and checks the'should be' bits of the encoding, as described in *Instruction encoding diagrams and pseudocode* on page AppxE-2.

E.7.21 ExceptionTaken()

This function is used with PushStack () for exception entry.

E.7.22 ExclusiveMonitorsPass()

This function determines whether a store exclusive instruction is successful. A store exclusive is successful if it still has possession of the exclusive monitors.

E.7.23 FindPriv()

Used to determine if privileged execution.

E.7.24 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option)
```

E.7.25 Hint_SendEvent()

This procedure performs a *send event* hint.

E.7.26 Hint_Yield()

This procedure performs a *Yield* hint.

E.7.27 InITBlock()

This function returns `TRUE` if execution is currently in an IT block and `FALSE` otherwise.

```
boolean InITBlock()
```

ARMv6-M does not support the IT block, see *Hint instructions* on page A5-10.

E.7.28 InstructionSynchronizationBarrier()

This procedure produces an Instruction Synchronization Barrier.

```
InstructionSynchronizationBarrier(bits(4) option)
```

E.7.29 IntegerZeroDivideTrappingEnabled()

This function returns TRUE if the trapping of divisions by zero in the integer division instructions `SDIV` and `UDIV` is enabled, and FALSE otherwise.

In the M profile, this is controlled by the DIV_0_TRP bit in the Configuration Control register. TRUE is returned if the bit is 1 and FALSE if it is 0.

E.7.30 IsAligned()

Used for alignment checking in a memory access.

E.7.31 LastInITBlock()

This function returns TRUE if the current instruction is the last instruction in an IT block, and FALSE otherwise.

ARMv6-M does not support the IT block, see *Hint instructions* on page A5-10.

E.7.32 LoadWritePC()

This procedure writes a value to the PC with `BX`-like interworking behavior for writes by load instructions.

```
LoadWritePC(bits(32) value)
```

Note

The M profile only supports the Thumb execution state. An attempt to change the instruction execution state causes an exception.

E.7.33 MarkExclusiveGlobal()

Set the global monitor associated with ProcessorID() to the Exclusive Access state and save a tag address.

E.7.34 MarkExclusiveLocal()

Set the local monitor associated with ProcessorID() to the Exclusive Access state. it is IMPLEMENTATION DEFINED whether an address tag is saved.

E.7.35 Mem*[]

The following note relates to all the memory access helper functions:

- MemA[]
- MemA_unpriv[].
- MemU[]
- MemU_unpriv[].

Note

The ARM architecture does not mandate the nature of the interface to memory or the signalling of memory transactions on this interface. The interface details are IMPLEMENTATION DEFINED, within the constraints of the ARM architecture as described in Chapter A3 *Application Level Memory Model*.

E.7.36 MemA[]

This array-like function performs a memory access that is required to be aligned, using the current privilege level.

E.7.37 MemA_unpriv[]

This array-like function performs a memory access that is required to be aligned, as an unprivileged access regardless of the current privilege level.

E.7.38 MemU[]

This array-like function performs a memory access that is allowed to be unaligned, using the current privilege level.

E.7.39 MemU_unpriv[]

This array-like function performs a memory access that is allowed to be unaligned, as an unprivileged access regardless of the current privilege level.

E.7.40 PopStack()

This function is used to recover context from the stack on exception return.

E.7.41 ProcessorID()

Identifies the executing processor.

E.7.42 PushStack()

This function is used to store context onto the stack during exception entry.

E.7.43 R[]

This array-like function reads or writes a register. Reading register 13, 14, or 15 reads the SP, LR or PC respectively and writing register 13 or 14 writes the SP or LR respectively.

```
bits(32) R[integer n]
R[integer n] = bits(32) value;
```

E.7.44 RaiseIntegerZeroDivide()

This procedure raises the appropriate exception for a division by zero in the integer division instructions `SDIV` and `UDIV`.

In the M profile, this is a UsageFault exception.

E.7.45 SetExclusiveMonitors()

This procedure sets the exclusive monitors for a load exclusive instruction.

E.7.46 SetPending()

This procedure sets the associated exception state to Pending. For a definition of the different exception states see *Exceptions* on page B1-4.

E.7.47 Shift(), Shift_C()

These functions perform standard ARM shifts on values, returning a result value and in the case of `Shift_C()`, a carry out bit. See *Shift operations* on page A6-10.

E.7.48 ThisInstr()

This function returns the currently-executing instruction. It is only used on 32-bit instruction encodings at present.

```
bits(32) ThisInstr()
```

E.7.49 ThumbExpandImm(), ThumbExpandImmWithC()

These functions do the standard expansion of the 12 bits specifying an Thumb data-processing immediate to its 32-bit value. The `WithC` version also produces a carry out bit. Not supported in ARMv6-M.

E.7.50 ValidateAddress()

Determine permissions and memory attributes associated with the given address as part of a memory access, and generate an access violation exception as appropriate.

Appendix F
Pseudocode Index

This appendix provides an index to pseudocode operators and functions that occur elsewhere in the document. It contains the following sections:

- *Pseudocode operators and keywords* on page AppxF-2
- *Pseudocode functions and procedures* on page AppxF-5.

F.1 Pseudocode operators and keywords

Table F-1 lists the pseudocode operators and keywords, and is an index to their descriptions:

Table F-1 Pseudocode operators and keywords

Operator	Meaning	See
`-`	Unary minus on integers or reals	*Unary plus, minus and absolute value* on page AppxE-15
`-`	Subtraction of integers, reals and bitstrings	*Addition and subtraction* on page AppxE-15
`+`	Unary plus on integers or reals	*Unary plus, minus and absolute value* on page AppxE-15
`+`	Addition of integers, reals and bitstrings	*Addition and subtraction* on page AppxE-15
`(...)`	Around arguments of procedure	*Procedure calls* on page AppxE-17, *Procedure and function definitions* on page AppxE-21
`(...)`	Around arguments of function	*General expression syntax* on page AppxE-9, *Procedure and function definitions* on page AppxE-21
`.`	Extract named member from a list	*Lists* on page AppxE-7
`.`	Extract named bit or field from a register	*Bitstring extraction* on page AppxE-12
`!`	Boolean NOT	*Operations on booleans* on page AppxE-11
`!=`	Compare for non-equality (any type)	*Equality and non-equality testing* on page AppxE-11
`!=`	Compare for non-equality (between integers and reals)	*Comparisons* on page AppxE-15
`&&`	Boolean AND	*Operations on booleans* on page AppxE-11
`*`	Multiplication of integers and reals	*Multiplication* on page AppxE-15
`/`	Division of integers and reals (real result)	*Division and modulo* on page AppxE-15
`/*...*/`	Comment delimiters	*Comments* on page AppxE-21
`//`	Introduces comment terminated by end of line	*Comments* on page AppxE-21
`:`	Bitstring concatenation	*Bitstring concatenation and replication* on page AppxE-12
`:`	Integer range in bitstring extraction operator	*Bitstring extraction* on page AppxE-12
`[...]`	Around array index	*Arrays* on page AppxE-8

Table F-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
`[...]`	Around arguments of array-like function	*General expression syntax* on page AppxE-9, *Procedure and function definitions* on page AppxE-21
`^`	Boolean exclusive-OR	*Operations on booleans* on page AppxE-11
`||`	Boolean OR	*Operations on booleans* on page AppxE-11
`<`	*Less than* comparison of integers and reals	*Comparisons* on page AppxE-15
`<...>`	Extraction of specified bits of bitstring or integer	*Bitstring extraction* on page AppxE-12
`<<`	Multiply integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxE-16
`<=`	*Less than or equal* comparison of integers and reals	*Comparisons* on page AppxE-15
`=`	Assignment	*Assignments* on page AppxE-17
`==`	Compare for equality (any type)	*Equality and non-equality testing* on page AppxE-11
`==`	Compare for equality (between integers and reals)	*Comparisons* on page AppxE-15
`>`	*Greater than* comparison of integers and reals	*Comparisons* on page AppxE-15
`>=`	*Greater than or equal* comparison of integers and reals	*Comparisons* on page AppxE-15
`>>`	Divide integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxE-16
`2^N`	Power of two (real result)	*Scaling* on page AppxE-16
`AND`	Bitwise AND of bitstrings	*Logical operations on bitstrings* on page AppxE-13
`array`	Keyword introducing array type definition	*Arrays* on page AppxE-8
`bit`	Bitstring type of length 1	*Bitstrings* on page AppxE-5
`bits(N)`	Bitstring type of length N	*Bitstrings* on page AppxE-5
`boolean`	Boolean type	*Booleans* on page AppxE-6
`case ... of ...`	Control structure	*case ... of ...* on page AppxE-20
`DIV`	Quotient from integer division	*Division and modulo* on page AppxE-15
`enumeration`	Keyword introducing enumeration type definition	*Enumerations* on page AppxE-6

Table F-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
`EOR`	Bitwise EOR of bitstrings	*Logical operations on bitstrings* on page AppxE-13
`FALSE`	Boolean constant	*Booleans* on page AppxE-6
`for ...`	Control structure	*for ...* on page AppxE-20
`if ... then ... else ...`	Expression selecting between two values	*Conditional selection* on page AppxE-11
`if ... then ... else ...`	Control structure	*if ... then ... else ...* on page AppxE-18
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior	*IMPLEMENTATION_DEFINED* on page AppxE-18
`integer`	Unbounded integer type	*Integers* on page AppxE-6
`MOD`	Remainder from integer division	*Division and modulo* on page AppxE-15
`OR`	Bitwise OR of bitstrings	*Logical operations on bitstrings* on page AppxE-13
`otherwise`	Introduces default case in `case ... of ...` control structure	*case ... of ...* on page AppxE-20
`real`	Real number type	*Reals* on page AppxE-6
`repeat ... until ...`	Control structure	*repeat ... until ...* on page AppxE-19
`return`	Procedure or function return	*Return statements* on page AppxE-17
`SEE`	Points to other pseudocode to use instead	*SEE...* on page AppxE-18
`SUBARCHITECTURE_DEFINED`	Describes SUBARCHITECTURE DEFINED behavior	*SUBARCHITECTURE_DEFINED* on page AppxE-18
`TRUE`	Boolean constant	*Booleans* on page AppxE-6
`UNDEFINED`	Cause Undefined Instruction exception	*UNDEFINED* on page AppxE-17
`UNKNOWN`	Unspecified value	*General expression syntax* on page AppxE-9
`UNPREDICTABLE`	Unspecified behavior	*UNPREDICTABLE* on page AppxE-17
`when`	Introduces specific case in `case ... of ...` control structure	*case ... of ...* on page AppxE-20
`while ... do ...`	Control structure	*while ... do* on page AppxE-20

F.2 Pseudocode functions and procedures

Table F-2 lists the pseudocode functions and procedures used in this manual, and is an index to their descriptions:

Table F-2 Pseudocode functions and procedures

Function	Meaning	See
`_Mem[]`	Basic memory accesses	on page B2-6*The _Mem[] function* on page B2-6
`Abs()`	Absolute value of an integer or real	*Unary plus, minus and absolute value* on page AppxE-15
`AddWithCarry()`	Addition of bitstrings, with carry input and carry/overflow outputs	*Pseudocode details of addition and subtraction* on page A2-8
`Align()`	Align integer or bitstring to multiple of an integer	*Rounding and aligning* on page AppxE-16
`ALUWritePC()`	Write value to PC, with interworking for ARM only from ARMv7	*Pseudocode details of ARM core register operations* on page A2-11
`ArchVersion()`	Major version number of the architecture	*ArchVersion()* on page AppxE-22
`ASR()`	Arithmetic shift right of a bitstring	*Shift and rotate operations* on page A2-5
`ASR_C()`	Arithmetic shift right of a bitstring, with carry output	*Shift and rotate operations* on page A2-5
`BadReg()`	Test for register number 13 or 15	*BadReg()* on page AppxE-22
`BigEndianReverse()`	Endian-reverse the bytes of a bitstring	*Declarations and support functions* on page B2-3
`BitCount()`	Count number of ones in a bitstring	*Bitstring count* on page AppxE-13
`BranchTo()`	Continue execution at specified address	*Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-11
`BranchWritePC()`	Write value to PC, without interworking	*Pseudocode details of ARM core register operations* on page A2-11
`BXWritePC()`	Write value to PC, with interworking	
`CallSupervisor()`	Generate exception for SVC instruction	*CallSupervisor()* on page AppxE-23
`ClearEventRegister()`	Clear the Event Register of the current processor	*Pseudocode details of the Wait For Event lock mechanism* on page B1-35
`ClearExclusiveByAddress()`	Clear global exclusive monitor records for an address range.	See note associated with MemA[].
`ConditionPassed()`	Returns TRUE if the current instruction passes its condition check	*Pseudocode details of conditional execution* on page A6-9
`Consistent()`	Test identically-named instruction bits or fields are identical	*Instruction encoding diagrams and pseudocode* on page AppxE-2

Table F-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`CountLeadingSignBits()`	Number of identical sign bits at left end of bitstring, excluding the leftmost bit itself	*Lowest and highest set bits of a bitstring* on page AppxE-13
`CountLeadingZeroBits()`	Number of zeros at left end of bitstring	
`CurrentCond()`	Returns condition for current instruction	*Pseudocode details of conditional execution* on page A6-9
`DataAddressMatch()`	DWT comparator data address matching	*Comparator support - data address matching* on page C1-21
`DataMemoryBarrier()`	Perform a Data Memory Barrier operation	*DataMemoryBarrier()* on page AppxE-23
`DataSynchronizationBarrier()`	Perform a Data Synchronization Barrier operation	*DataSynchronizationBarrier()* on page AppxE-24
`Deactivate()`	Removal of Active state from an exception as part of the exception return	*Exception return behavior* on page B1-21
`DecodeImmShift()`	Decode shift type and amount for an immediate shift	*Shift operations* on page A6-10
`DecodeRegShift()`	Decode shift type for a register-controlled shift	*Shift operations* on page A6-10
`DefaultAttributes()`	Determine memory attributes for an address in the default memory map	*Memory attribute definitions and mapping* on page B2-4
`Edge()`	Edge detection for external interrupts	*External interrupt input behavior* on page B3-18
`EncodingSpecificOperations()`	Invoke encoding-specific pseudocode and *should be* checks	*Instruction encoding diagrams and pseudocode* on page AppxE-2
`EventRegistered()`	Determine whether the Event Register of the current processor is set	*Pseudocode details of the Wait For Event lock mechanism* on page B1-35
`ExceptionEntry()`	Exception entry behavior	*Exception entry behavior* on page B1-18
`ExceptionIN()`	Determine exception entry status	*External interrupt input behavior* on page B3-18
`ExceptionOUT()`	Determine exception return status	*External interrupt input behavior* on page B3-18
`ExceptionReturn()`	Exception return behavior	*Exception return behavior* on page B1-21
`ExceptionTaken()`	Part of `ExceptionEntry()` behavior	*Exception entry behavior* on page B1-18
`FindPriv()`	Determine access privilege	*Declarations and support functions* on page B2-3
`HighestSetBit()`	Position of leftmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxE-13

Table F-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`Hint_Debug()`	Perform function of DBG hint instruction	*Hint_Debug()* on page AppxE-24
`Hint_SendEvent()`	Perform function of SEV hint instruction	*Hint_SendEvent()* on page AppxE-25
`Hint_Yield()`	Perform function of YIELD hint instruction	*Hint_Yield()* on page AppxE-25
`InITBlock()`	Return TRUE if current instruction is in an IT block. Always FALSE in ARMv6-M.	*Conditional execution* on page A6-8
`InterruptAssertion()`	Determine status of an external interrupt	*External interrupt input behavior* on page B3-18
`InstrAddressMatch()`	DWT comparator instruction address matching	*Comparator support - instruction address matching* on page C1-20
`InstructionSynchronizationBarrier()`	Perform an Instruction Synchronization Barrier operation	*InstructionSynchronizationBarrier()* on page AppxE-25
`Int()`	Convert bitstring to integer in argument-specified fashion	*Converting bitstrings to integers* on page AppxE-14
`IntegerZeroDivideTrappingEnabled()`	Check whether divide-by-zero trapping is enabled for integer divide instructions	*IntegerZeroDivideTrappingEnabled()* on page AppxE-25
`IsAligned()`	Address alignment check	*Declarations and support functions* on page B2-3
`IsOnes()`	Test for all-ones bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxE-13
`IsOnesBit()`	Test for all-ones bitstring (bit result)	
`IsZero()`	Test for all-zeros bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxE-13
`IsZeroBit()`	Test for all-zeros bitstring (bit result)	
`LastInITBlock()`	Return TRUE if current instruction is the last instruction in an IT block. Always FALSE in ARMv6-M.	*LastInITBlock()* on page AppxE-25
`LateArrival()`	Late arrival exception handling	*Late arriving exceptions* on page B1-25
`Len()`	Bitstring length	*Bitstring length and top bit* on page AppxE-12
`LoadWritePC()`	Write value to PC, with interworking (without it before ARMv5T)	*Pseudocode details of ARM core register operations* on page A2-11
`LookUpSP()`	Select the current SP	*Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-11
`LowestSetBit()`	Position of rightmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxE-13

Table F-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`LSL()`	Logical shift left of a bitstring	*Shift and rotate operations* on page A2-5
`LSL_C()`	Logical shift left of a bitstring, with carry output	
`LSR()`	Logical shift right of a bitstring	
`LSR_C()`	Logical shift right of a bitstring, with carry output	
`Max()`	Maximum of integers or reals	*Maximum and minimum* on page AppxE-16
`MemA[]`	Memory access that must be aligned, at current privilege level	*The MemA[] function* on page B2-6
`MemoryAccessAddress()`	Combine attributes and address for memory access	*Memory attribute definitions and mapping* on page B2-4
`MemU[]`	Memory access without alignment requirement, at current privilege level	*Memory accesses* on page B2-6
`Min()`	Minimum of integers or reals	*Maximum and minimum* on page AppxE-16
`NOT()`	Bitwise inversion of a bitstring	*Logical operations on bitstrings* on page AppxE-13
`Ones()`	All-ones bitstring	*Bitstring concatenation and replication* on page AppxE-12
`ProcessorID()`	Return integer identifying the processor	*ProcessorID()* on page AppxE-27
`PopStack()`	Stack restore sequence on an exception return	*Exception return behavior* on page B1-21
`PushStack()`	Stack save sequence on exception entry	*Exception entry behavior* on page B1-18
`R[]`	Access the main ARM core register bank	*Pseudocode details of ARM core register operations* on page A2-11 *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-11
`Replicate()`	Bitstring replication	*Bitstring concatenation and replication* on page AppxE-12
`ReturnAddress()`	Return address stacked on exception entry	*Exception entry behavior* on page B1-18

Table F-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`ROR()`	Rotate right of a bitstring	*Shift and rotate operations* on page A2-5
`ROR_C()`	Rotate right of a bitstring, with carry output	
`RRX()`	Rotate right with extend of a bitstring	
`RRX_C()`	Rotate right with extend of a bitstring, with carry output	
`Sat()`	Convert integer to bitstring with specified saturation	*Pseudocode details of saturation* on page A2-9
`SatQ()`	Convert integer to bitstring with specified saturation, with saturated flag output	
`SendEvent()`	Create a WFE wake up event that sets the Event Register(s) on execution of an `SEV` instruction.	*Pseudocode details of the Wait For Event lock mechanism* on page B1-35
`SetEventRegister()`	Set the Event Register of the current processor	*Pseudocode details of the Wait For Event lock mechanism* on page B1-35
`Shift()`	Perform a specified shift by a specified amount on a bitstring	*Shift operations* on page A6-10
`Shift_C()`	Perform a specified shift by a specified amount on a bitstring, with carry output	
`SignedSat()`	Convert integer to bitstring with signed saturation	*Pseudocode details of saturation* on page A2-9
`SignedSatQ()`	Convert integer to bitstring with signed saturation, with saturated flag output	
`SignExtend()`	Extend bitstring to left with copies of its leftmost bit	*Zero-extension and sign-extension of bitstrings* on page AppxE-13
`SInt()`	Convert bitstring to integer in signed (two's complement) fashion	*Converting bitstrings to integers* on page AppxE-14
`TailChain()`	Tail chaining exception behavior	*Tail-chaining* on page B1-27
`TakeReset()`	Reset behavior	*Reset behavior* on page B1-17
`ThisInstr()`	Returns the bitstring encoding of the current instruction	*ThisInstr()* on page AppxE-28
`TopBit()`	Leftmost bit of a bitstring	*Bitstring length and top bit* on page AppxE-12
`UInt()`	Convert bitstring to integer in unsigned fashion	*Converting bitstrings to integers* on page AppxE-14

Table F-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`UnsignedSat()`	Convert integer to bitstring with unsigned saturation	*Pseudocode details of saturation* on page A2-9
`UnsignedSatQ()`	Convert integer to bitstring with unsigned saturation, with saturated flag output	
`WaitForEvent()`	Wait until `WFE` instruction completes	*Pseudocode details of the Wait For Event lock mechanism* on page B1-35
`WaitForInterrupt()`	Wait until `WFI` instruction completes	*Pseudocode details of Wait For Interrupt* on page B1-36
`WriteToRegField()`	Indicate a write of '1' to a specified field in a system control register	*External interrupt input behavior* on page B3-18
`ZeroExtend()`	Extend bitstring to left with zero bits	*Zero-extension and sign-extension of bitstrings* on page AppxE-13
`Zeros()`	All-zeros bitstring	*Bitstring concatenation and replication* on page AppxE-12

Appendix G
Register Index

This appendix provides an index to the descriptions of the ARM registers (core and memory mapped) in the document. It contains the following sections:

G.1 ARM core registers

Table G-1 provides an index to the main descriptions of the ARM core registers defined in ARMv6-M.

Table G-1 ARM core register index

Register	Description, see
R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12	*Registers* on page B1-8
SP_main, SP_process	*The SP registers* on page B1-8
LR (R14)	*Registers* on page B1-8
PC (R15)	*Registers* on page B1-8
APSR[a]	*The special-purpose program status registers (xPSR)* on page B1-8
IPSR[a]	*The special-purpose program status registers (xPSR)* on page B1-8
EPSR[ab]	*The special-purpose program status registers (xPSR)* on page B1-8
PRIMASK	*The special-purpose mask register* on page B1-10
CONTROL	*The special-purpose control register* on page B1-10

a. xPSR = APSR | IPSR | EPSR
b. EPSR [24] is known as the T-bit (bit tbit)

G.2 Memory mapped system registers

Table G-2 provides an index to the main descriptions of the memory mapped system control registers defined in ARMv6-M. The registers are listed in the order they are described in this manual.

Table G-2 memory-mapped control register index

Register	Description, see
CPUID	*CPUID Base Register – (CPUID, 0xE000ED00)* on page B3-7
ICSR	*Interrupt Control and State Register – (0xE000ED04)* on page B3-8
AIRCR	*Application Interrupt and Reset Control Register – (0xE000ED0C)* on page B3-9
CCR	*Configuration and Control Register – (0xE000ED14)* on page B3-11
SHPR2	*System Handler Priority Register 2 – (0xE000ED1C)* on page B3-11
SHPR3	*System Handler Priority Register 3 – (0xE000ED20)* on page B3-11
SHCSR	*System Handler Control and State Register – (0xE000ED24)* on page B3-12
~~ST~~CSR	*SysTick Control and Status Register – (0xE000E010)* on page B3-15
~~ST~~RVR	*SysTick Reload Value Register – (0xE000E014)* on page B3-15
~~ST~~CVR	*SysTick Current Value Register – (0xE000E018)* on page B3-16
~~STCR~~	*SysTick Calibration Value Register – (0xE000E01C)* on page B3-16
ISER	*Interrupt Set-Enable Register – (0xE000E100)* on page B3-20
ICER	*Interrupt Clear-Enable Register – (0xE000E180)* on page B3-20
ISPR	*Interrupt Set-Pending Register – (0xE000E200)* on page B3-20
ICPR	*Interrupt Clear-Pending Register – (0xE000E280)* on page B3-21
IPR[x][a]	*Interrupt Priority Registers – (0xE000E400-E41C)* on page B3-21

a. 4 priority fields per register, with an architecture requirement of a priority field per interrupt supported.

G.3 Memory mapped debug registers

Table G-3 provides an index to the main descriptions of the memory mapped debug registers defined in the ARMv6-M Debug Extension. The registers are listed in the order they are described in this manual.

Table G-3 memory-mapped debug register index

Register[a]	Description, see
DFSR	*Debug Fault Status Register (0xE000ED30)* on page C1-12
DHCSR	*Debug Halting Control and Status Register – (0xE000EDF0)* on page C1-13
DCRSR	*Debug Core Register Selector Register – (0xE000EDF4)* on page C1-15
DCRDR	*Debug Core Register Data Register – (0xE000EDF8)* on page C1-16
DEMCR	*Debug Exception and Monitor Control Register – (0xE000EDFC)* on page C1-17
DWT_CTRL	*DWT_CTRL (0xE0001000)* on page C1-23
DWT_PCSR	*DWT_PCSR (0xE000101C)* on page C1-23
DWT_COMPx	*DWT_COMPx (0xE0001020 etc.)* on page C1-24
DWT_MASKx	*DWT_MASKx (0xE0001024 etc.)* on page C1-24
DWT_FUNCTIONx	*DWT_FUNCTIONx (0xE0001028 etc.)* on page C1-25
BP_CTRL	*BP_CTRL (0xE0002000)* on page C1-27
BP_COMPx	*BP_COMPx breakpoint comparison (0xE0002008-2040)* on page C1-28

a. In addition to the registers listed, the Debug Extension includes bits in the ICSR, see *Interrupt Control State Register (ICSR)* on page B3-8.

Glossary

AAPCS

Procedure Call Standard for the ARM Architecture.

Addressing mode

Means a method for generating the memory address used by a load/store instruction.

Aligned

Refers to data items stored in such a way that their address is divisible by the highest power of 2 that divides their size. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of an element of the access

APSR

See Application Program Status Register.

Application Program Status Register

The register containing those bits that deliver status information about the results of instructions, the N, Z, C, and V bits of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Atomicity

Is a term that describes either single-copy atomicity or multi-copy atomicity. The forms of atomicity used in the ARM architecture are defined in *Atomicity* on page A3-9.

See also Multi-copy Atomicity, Single-copy atomicity.

Banked register

Is a register that has multiple instances, with the instance that is in use depending on the processor mode, security state, or other processor state.

Base register
Is a register specified by a load/store instruction that is used as the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register write-back
Describes writing back a modified value to the base register used in an address calculation.

Big-endian memory
Means that:

- a byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the most significant byte in the halfword at that address.

Blocking
Describes an operation that does not permit following instructions to be executed before the operation is completed.

A non-blocking operation can permit following instructions to be executed before the operation is completed, and in the event of encountering an exception do not signal an exception to the core. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise processor state.

Branch prediction
Is where a processor chooses a future execution path to prefetch along (see Prefetching). For example, after a branch instruction, the processor can choose to prefetch either the instruction following the branch or the instruction at the branch target.

Breakpoint
Is a debug event triggered by the execution of a particular instruction, specified in terms of the address of the instruction and/or the state of the processor when the instruction is executed.

Byte Is an 8-bit data item.

Cache Is a block of high-speed memory locations whose addresses are changed automatically in response to which memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.

Cache contention
Is when the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity goes up and performance drops.

Cache hit
Is a memory access that can be processed at high speed because the data it addresses is already in the cache.

Cache line
Is the basic unit of storage in a cache. Its size is always a power of two (usually 4 or 8 words), and must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely just called cache lines.

Cache miss
Is a memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Callee-save registers
Are registers that a called procedure must preserve. To preserve a callee-save register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and re-load it from the stack during procedure exit.

Caller-save registers
Are registers that a called procedure need not preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

Clear Relates to registers or register fields. Indicates the bit has a value of zero (or bit field all 0s), or is being written with zero or all 0s.

Conditional execution
Means that if the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Configuration
Settings made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

Context switch
Is the saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch is used to describe any situations where the context is switched by an operating system and might or might not include changes to the address space.

DCB Debug Control Block - a region within the System Control Space (see SCS) specifically assigned to register support of debug features.

Digital signal processing (DSP)
Refers to a variety of algorithms that are used to process signals that have been sampled and converted to digital form. Saturated arithmetic is often used in such algorithms.

Direct Memory Access
Is an operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Do-not-modify fields (DNM)
Means the value must not be altered by software. DNM fields read as UNKNOWN values, and can only be written with the same value read from the same field on the same processor.

Doubleword
Is a 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.

Doubleword-aligned
Means that the address is divisible by 8.

DSP *See* Digital signal processing

DWT Data Watchpoint and Trace - part of the ARM debug architecture.

Endianness
Is an aspect of the system's memory mapping. See big-endian and little-endian.

EPSR *See* Execution Program Status Register.

ETM Embedded Trace Macrocell - part of the ARM debug architecture

Exception
Handles an event. For example, an exception could handle an external interrupt or an Undefined Instruction.

Exception vector
Is one of a number of fixed addresses in low memory, or in high memory if high vectors are configured.

Execution Program Status Register
The register that contains the execution state bits and is part of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Execution stream
The stream of instructions that would have been executed by sequential execution of the program.

Explicit access
A read from memory, or a write to memory, generated by a load or store instruction executed in the processor. Reads and writes generated by L1 DMA accesses or hardware translation table accesses are not explicit accesses.

Fault An exception due to some form of system error.

General-purpose register
Is one of the 32-bit general-purpose integer registers, R0 to R15. Note that R15 holds the Program Counter, and there are often limitations on its use that do not apply to R0 to R14.

Halfword
Is a 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

Halfword-aligned
Means that the address is divisible by 2.

High registers
Are ARM core registers 8 to 15, that can be accessed by some Thumb instructions.

Immediate and offset fields
Are unsigned unless otherwise stated.

Immediate values
Are values that are encoded directly in the instruction and used as numeric data when the instruction is executed. Many ARM and Thumb instructions permit small numeric values to be encoded as immediate values in the instruction that operates on them.

IMP Is an abbreviation used in diagrams to indicate that the bit or bits concerned have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

Index register
Is a register specified in some load/store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some addressing modes optionally permit the index register value to be shifted before the addition or subtraction.

Inline literals
These are constant addresses and other data items held in the same area as the code itself. They are automatically generated by compilers, and can also appear in assembler code.

Interrupt Program Status Register
The register that provides status information on whether an application thread or exception handler is currently executing on the processor. If an exception handler is executing, the register provides information on the exception type. The register is part of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Interworking
Is a method of working that permits branches between ARM and Thumb code in architecture variants that support both execution states.

IPSR *See* Interrupt Program Status Register.

ITM Instrumentation Trace Macrocell - part of the ARM debug architecture

Little-endian memory
Means that:

- a byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the least significant byte in the halfword at that address.

Load/Store architecture
Is an architecture where data-processing operations only operate on register contents, not directly on memory contents.

Long branch
Is the use of a load instruction to branch to anywhere in the 4GB address space.

Memory barrier
See *Memory barriers* on page A3-19.

Memory coherency
Is the problem of ensuring that when a memory location is read (either by a data read or an instruction fetch), the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory, a write buffer and/or cache(s).

Memory hint

A memory hint instruction allows you to provide advance information to memory systems about future memory accesses, without actually loading or storing any data to or from the register file.

Memory-mapped I/O

Uses special memory addresses that supply I/O functions when they are loaded from or stored to.

Memory Protection Unit (MPU)

Is a hardware unit whose registers provide simple control of a limited number of protection regions in memory.

MPU *See* Memory Protection Unit.

NRZ Non-Return-to-Zero - physical layer signalling scheme used on asynchronous communication ports.

Multi-copy atomicity

Is the form of atomicity described in *Multi-copy atomicity* on page A3-10.

See also Atomicity, Single-copy atomicity.

Offset addressing

Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

Physical address

Identifies a main memory location.

Post-indexed addressing

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Prefetching

Is the process of fetching instructions from memory before the instructions that precede them have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Pre-indexed addressing

Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Privileged access

Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

Protection region

Is a memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit

See Memory Protection Unit.

Pseudo-instruction

UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>,<Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>,<Rm>,#<n>`

PSR Program Status Register. *See* APSR, EPSR, IPSR and xPSR.

RAZ *See* Read-As-Zero fields.

RAO/SBOP field

Read-As-One, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1 (or all 1s for a bit field), and writes to the field must be ignored.

Software can rely on the field reading as 1 (or all 1s), but must use an SBOP policy to write to the field.

RAZ/SBZP field

Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0 (or all 0s for a bit field), and writes to the field must be ignored.

Software can rely on the field reading as zero, but must use an SBZP policy to write to the field.

Read-As-Zero fields (RAZ)

Appear as zero when read.

Read-Modify-Write fields (RMW)

Are read to a general-purpose register, the relevant fields updated in the register, and the register value written back.

Reserved

Unless otherwise stated:

- instructions that are reserved or that access reserved registers have UNPREDICTABLE behavior
- bit positions described as Reserved are UNK/SBZP.

Return Link

a value relating to the return address

R/W1C register bits marked R/W1C can be read normally and support write-one-to-clear. A read then write of the result back to the register will clear all bits set. R/W1C protects against read-modify-write errors occurring on bits set between reading the register and writing the value back (since they are written as zero, they will not be cleared).

RAZ/WI Relates to registers or register fields. Read as zero, ignore writes. RAZ can be used on its own.

RO Read only register or register field. RO bits are ignored on write accesses.

RISC Reduced Instruction Set Computer.

RMW *See* Read-Modify-Write fields.

Rounding error

Is defined to be the value of the rounded result of an arithmetic operation minus the exact result of the operation.

Saturated arithmetic

Is integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in ARM processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.

SBO *See* Should-Be-One fields.

SBOP *See* Should-Be-One-or-Preserved fields.

SBZ *See* Should-Be-Zero fields.

SBZP *See* Should-Be-Zero-or-Preserved fields.

SCB System Control Block - an address region within the System Control Space used for key feature control and configuration associated with the exception model.

SCS System Control Space - a 4kB region of the memory map reserved for system control and configuration.

Security hole

Is a mechanism that bypasses system protection.

Set Relates to registers or register fields. Indicates the bit has a value of 1 (or bit field all 1s), or is being written with 1 or all 1s, unless explicitly stated otherwise.

SWO Serial Wire Output - an asynchronous TPIU port supporting NRZ and/or Manchester encoding.

SWV Serial Wire Viewer - the combination of an SWO and DWT/ITM data tracing capability

Self-modifying code

Is code that writes one or more instructions to memory and then executes them. This type of code cannot be relied on without the use of barrier instructions to ensure synchronization.

Should-Be-One fields (SBO)

Should be written as 1 (or all 1s for a bit field) by software. Values other than 1 produce UNPREDICTABLE results.

Should-Be-One-or-Preserved fields (SBOP)

Should be written as 1 (or all 1s for a bit field) by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read, the value in the field should be preserved by writing the same value that has been previously read from the same field on the same processor.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Should-Be-Zero fields (SBZ)

Should be written as 0 (or all 0s for a bit field) by software. Values other than 0 produce UNPREDICTABLE results.

Should-Be-Zero-or-Preserved fields (SBZP)

Should be written as 0 (or all 0s for a bit field) by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read, the value in the field should be preserved by writing the same value that has been previously read from the same field on the same processor.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Signed data types

Represent an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

Signed immediate and offset fields

Are encoded in two’s complement notation unless otherwise stated.

SIMD Means Single-Instruction, Multiple-Data operations.

Single-copy atomicity

Is the form of atomicity described in *Single-copy atomicity* on page A3-9.

See also Atomicity, Multi-copy atomicity.

Spatial locality

Is the observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

SUBARCHITECTURE DEFINED

Means that the behavior is expected to be specified by a subarchitecture definition. Typically, this will be shared by multiple implementations, but it must only be relied on by specified types of code. This minimizes the software changes required when a new subarchitecture has to be developed.

SVC Is a supervisor call.

SWI Is a former term for SVC.

Status registers

See APSR, EPSR, IPSR and xPSR.

Temporal locality

Is the observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction

Is one or two halfwords that specify an operation for a processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

TPIU Trace Port Interface Unit - part of the ARM debug architecture

UAL *See* Unified Assembler Language.

Unaligned

An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses

Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated

Except where otherwise stated, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as UNDEFINED, UNPREDICTABLE, or an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

UNDEFINED

Indicates an instruction that generates an Undefined Instruction exception.

Unified Assembler Language

The assembler language introduced with Thumb-2 technology and used in this document. See *Unified Assembler Language* on page A4-4 for details.

Unified cache

Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing

Means addressing in which the base register value is used directly as the address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNK/SBOP field

UNKNOWN on reads, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1 (or all 1s for a bit field), and writes to the field must be ignored.

Software must not rely on the field reading as 1 (or all 1s), and must use an SBOP policy to write to the field.

UNK/SBZP field

UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0 (or all 0s for a bit field), and writes to the field must be ignored.

Software must not rely on the field reading as zero, and must use an SBZP policy to write to the field.

UNK field
Contains an UNKNOWN value.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

Unsigned data types
Represent a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.

Watchpoint
Is a debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Word Is a 32-bit data item. Words are normally word-aligned in ARM systems.

WO Write only register or register field. WO bits are UNKNOWN on read accesses.

Word-aligned
Means that the address is divisible by 4.

Write buffer
Is a block of high-speed memory whose purpose is to optimize stores to main memory.

WYSIWYG
What You See Is What You Get, an acronym for describing predictable behavior of the output generated. Display to printed form and software source to executable code are examples of common use.

xPSR Is the term used to describe the combination of the APSR, EPSR and IPSR into a single 32-bit Program Status Register. See *The special-purpose program status registers (xPSR)* on page B1-8.